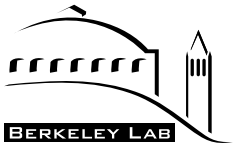


ERNEST ORLANDO LAWRENCE
BERKELEY NATIONAL LABORATORY

GenOpt(R)
Generic Optimization Program

User Manual
Version 2.0.0 β

Simulation Research Group
Building Technologies Department
Environmental Energy Technologies Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720

http://SimulationResearch.lbl.gov

Michael Wetter
MWetter@lbl.gov

June 23, 2003

Notice:

This work was supported by the U.S. Department of Energy (DOE), by the Swiss Academy of Engineering Sciences (SATW), and by the Swiss National Energy Fund (NEFF).

Contents

1 Abstract

GenOpt is an optimization program for the minimization of a cost function that is evaluated by an external simulation program. It has been developed for optimization problems where the cost function is computationally expensive and its derivatives are not available or may not even exist. GenOpt can be coupled to any simulation program that reads its input from text files and writes its output to text files. The independent variables can be continuous variables (possibly with lower and upper bounds), discrete variables, or both, continuous and discrete variables. Constraints on dependent variables can be implemented using penalty or barrier functions.

GenOpt has a library with local and global multi-dimensional and one-dimensional optimization algorithms, and algorithms for doing parametric runs. An algorithm interface allows adding new minimization algorithms without knowing the details of the program structure.

GenOpt is written in Java so that it is platform independent. The platform independence and the general interface make GenOpt applicable to a wide range of optimization problems.

GenOpt has not been designed for linear programming problems, quadratic programming problems, and problems where the gradient of the cost function is available. For such problems, as well as for other problems, special tailored software exist which is more efficient.

2 Notation

1. We use the notation $a \triangleq b$ to denote that a is equal to b by definition. We use the notation $a \leftarrow b$ to denote that a is assigned the value of b.
2. $\mathbb{R}^n$ denotes the Euclidean space of n-tuplets of real numbers. Vectors $x \in \mathbb{R}^n$ are always column vectors, and their elements are denoted by superscripts. The inner product in $\mathbb{R}^n$ is denoted by $\langle \cdot, \cdot \rangle$ and for $x, y \in \mathbb{R}^n$ defined by $\langle x, y \rangle \triangleq \sum_{i=1}^{n} x^i y^i$. The norm in $\mathbb{R}^n$ is denoted by $\| \cdot \|$ and for $x \in \mathbb{R}^n$ defined by $\|x\| \triangleq \langle x, x \rangle^{1/2}$.
3. We denote by $\mathbb{Z}$ the set of integers, by $\mathbb{Q}$ the set of rational numbers, and by $\mathbb{N} \triangleq \{0, 1, \ldots\}$ the set of natural numbers. The set $\mathbb{N}_+$ is defined as $\mathbb{N}_+ \triangleq \{1, 2, \ldots\}$. Similarly, vectors in $\mathbb{R}^n$ with strictly positive elements are denoted by $\mathbb{R}^n_+ \triangleq \{x \in \mathbb{R}^n \mid x^i > 0,\ i \in \{1, \ldots, n\}\}$ and the set $\mathbb{Q}_+$ is defined as $\mathbb{Q}_+ \triangleq \{q \in \mathbb{Q} \mid q > 0\}$.
4. Let $\mathbb{W}$ be a set containing a sequence $\{w_i\}_{i=0}^{k}$. Then, we denote by $\underline{w}_k$ the sequence $\{w_i\}_{i=0}^{k}$ and by $\underline{\mathbf{W}}_k$ the set of all $k+1$ element sequences in $\mathbb{W}$.
5. If $\mathbf{A}$ and $\mathbf{B}$ are sets, we denote by $\mathbf{A} \cup \mathbf{B}$ the union of $\mathbf{A}$ and $\mathbf{B}$ and by $\mathbf{A} \cap \mathbf{B}$ the intersection of $\mathbf{A}$ and $\mathbf{B}$.
6. If $\mathbf{S}$ is a set, we denote by $\overline{\mathbf{S}}$ the closure of $\mathbf{S}$ and by $2^{\mathbf{S}}$ the set of all nonempty subsets of $\mathbf{S}$.
7. If $\widehat{D} \in \mathbb{Q}^{n \times q}$ is a matrix, we will use the notation $\widehat{d} \in \widehat{D}$ to denote the fact that $\widehat{d} \in \mathbb{Q}^n$ is a column vector of the matrix $\widehat{D}$. Similarly, by $D \subset \widehat{D}$ we mean that $D \in \mathbb{Q}^{n \times p}$ $(1 \leq p \leq q)$ is a matrix containing only columns of $\widehat{D}$. Further, $\mathrm{card}(D)$ denotes the number of columns of D.
8. $f(\cdot)$ denotes a function where $(\cdot)$ stands for the undesignated variables. $f(x)$ denotes the value of $f(\cdot)$ at the point x. $f\colon A \to B$ indicates that the domain of $f(\cdot)$ is in the space A and its range in the space B.
9. We say that a function $f : \mathbb{R}^n \to \mathbb{R}$ is once continuously differentiable if $f(\cdot)$ is defined on $\mathbb{R}^n$, and if $f(\cdot)$ has continuous derivatives on $\mathbb{R}^n$.
10. For $x^* \in \mathbb{R}^n$ and $f\colon \mathbb{R}^n \to \mathbb{R}$ continuously differentiable, we say that x^* is stationary if $\nabla f(x^*) = 0$.
11. We denote by $\{e_i\}_{i=1}^{n}$ the unit vectors in $\mathbb{R}^n$.
12. We denote by $\rho \sim U(0,1)$ that $\rho \in \mathbb{R}$ is a uniformly distributed random number, with $0 \leq \rho \leq 1$.

3 Introduction

The use of system simulation for analyzing complex engineering problems is increasing. Such problems typically involve many independent variables[1], and can only be optimized by means of numerical optimization. Many designers use parametric studies to achieve better performance of such systems, even though such studies typically yield only partial improvement while requiring high labor time. In such parametric studies, one usually fixes all but one variable and tries to optimize a cost function[2] with respect to the non-fixed variable. The procedure is repeated iteratively by varying another variable. However, every time a variable is varied, all other variables typically become non-optimal and hence need also to be adjusted. It is clear that such a manual procedure is very time-consuming and often impractical for more than 2 or 3 independent variables.

GenOpt, a generic optimization program, has been developed to find with less labor time the independent variables that yield better performance of such systems. GenOpt does optimization of a user-supplied cost function, using a user-selected optimization algorithm.

In the most general form, the optimization problems addressed by GenOpt can be stated as follows: Let $\mathbf{X}$ be a user-specified constraint set, and let $f\colon \mathbf{X} \to \mathbb{R}$ be a user-defined cost function that is bounded from below. The constraint set $\mathbf{X}$ consists of all possible design options, and the cost function $f(\cdot)$ measures the system performance. GenOpt tries to find a solution to the problem[3]

$$\min_{x \in \mathbf{X}} f(x). \tag{3.1}$$

This problem is usually "solved" by iterative methods, which construct infinite sequences, of progressively better approximations to a "solution", i.e., a point that satisfies an optimality condition. If $\mathbf{X} \subset \mathbb{R}^n$, with some $n \in \mathbb{N}$, and $\mathbf{X}$ or $f(\cdot)$ is not convex, we do not have a test for global optimality, and the most one can obtain is a point that satisfies a local optimality condition. Furthermore, for $\mathbf{X} \subset \mathbb{R}^n$, tests for optimality are based on differentiability assumptions of the cost function. Consequently, optimization algorithms can fail, possibly far from a solution, if $f(\cdot)$ is not differentiable in the continuous

[1] The independent variables are the variables that are varied by the optimization algorithm from one iteration to the next. They are also called design parameters or free parameters.

[2] The cost function is the function being optimized. The cost function measures a quantity that should be minimized, such as a building's annual operation cost, a system's energy consumption, or a norm between simulated and measured values in a data fitting process. The cost function is also called objective function.

[3] If $f(\cdot)$ is discontinuous, it may only have an infimum (i.e., a greatest lower bound) but no minimum even if the constraint set $\mathbf{X}$ is compact. Thus, to be correct, (3.1) should be replaced by $\inf_{x \in \mathbf{X}} f(x)$. For simplicity, we will not make this distinction.

independent variables. Some optimization algorithms are more likely to fail at discontinuities than others. GenOpt has algorithms that are not very sensitive to (small) discontinuities in the cost function, such as Generalized Pattern Search algorithms, which can also be used in conjunction with heuristic global optimization algorithms.

Since one of GenOpt's main application fields is building energy use or operation cost optimization, GenOpt has been designed such that it addresses the special properties of optimization problems in this area. In particular, GenOpt is designed for optimization problems where

1. the cost function may have to be defined on approximate numerical solutions of differential algebraic equations, which may fail to be continuous (see Section 4.1.4),
2. the number of independent variables is small,[4]
3. evaluating the cost function requires much more computation time than determining the values for the next iterate,
4. analytical properties of the cost function (such as formula for the gradient) are not available.

GenOpt has following properties:

1. GenOpt can be coupled to any simulation program that calculates the cost function without having to modify or recompile either program, provided that the simulation program reads its input from text files and writes its output to text files.
2. The user can select an optimization algorithm from an algorithm library, or implement a custom algorithm without having to recompile and understand the whole optimization environment.
3. GenOpt does not require an expression for the gradient of the cost function.

With GenOpt, it is easy to couple a new simulation program, specify the optimization variables and minimize the cost function. Therefore, in designing complex systems, as well as in system analysis, a generic optimization program like GenOpt offers valuable assistance. Note, however, that optimization is not easy: The efficiency and success of an optimization is strongly affected by the properties and the formulation of the cost function, and by the selection of an appropriate optimization algorithm.

This manual is structured as follows: In Section 4, we classify optimization problems and discuss which of GenOpt's algorithms can be used for each of these problems. Next, we explain the algorithms that are implemented in GenOpt: In Section 5, we discuss the algorithms for multi-dimensional optimization; in Section 6 the algorithms for one-dimensional optimization; and

[4]By small, we mean in the order of 10, but the maximum number of independent variables is not restricted in GenOpt.

in Section 7 the algorithms for parametric runs. In Section 8, we discuss how constraints on independent variables are implemented, and how constraints on dependent variables can be implemented. In Section 9, we explain the structure of the GenOpt software, the interface for the simulation program and the interface for the optimization algorithms. How to install and start GenOpt is described in Section 10. Section 11 shows how to set up the configuration and input files, and how to use GenOpt's pre- and post-processing capabilities.

4 Optimization Problems

4.1 Classification of Optimization Problems

We will now classify some optimization problems that can be solved with GenOpt's optimization algorithms. The classification will be used in Section 4.2 to recommend suitable optimization algorithms.

We distinguish between problems whose design parameters are continuous variables[1], discrete variables[2], or both. In addition, we distinguish between problems with and without inequality constraints on the dependent variables.

4.1.1 Problems with Continuous Variables

We will use the notation

$$\mathbf{X} \triangleq \left\{ x \in \mathbb{R}^n \mid l^i \leq x^i \leq u^i,\ i \in \{1, \ldots, n\} \right\}, \tag{4.1}$$

where $-\infty \leq l^i < u^i \leq \infty$ for $i \in \{1, \ldots, n\}$, to denote box-constraints on independent continuous variables.

We will consider optimization problems of the form

$$\mathbf{P}_c \qquad \min_{x \in \mathbf{X}} f(x), \tag{4.2}$$

where $f \colon \mathbb{R}^n \to \mathbb{R}$ is a once continuously differentiable cost function.

Now, we add inequality constraints on the dependent variables to (4.2) and obtain

$$\mathbf{P}_{cg} \qquad \min_{x \in \mathbf{X}} f(x), \tag{4.3a}$$

$$g(x) \leq 0, \tag{4.3b}$$

where everything is as in (4.2) and, in addition, $g \colon \mathbb{R}^n \to \mathbb{R}^m$ is a once continuously differentiable constraint function (for some $m \in \mathbb{N}$). We will assume that there exists an $x^* \in \mathbf{X}$ that satisfies $g(x^*) < 0$.

4.1.2 Problems with Discrete Variables

Next, we will discuss the situation where all design parameters can only take on user-specified discrete values.

Let $\mathbf{X}_d \subset \mathbb{Z}^{n_d}$ denote the constraint set with a finite, non-zero number of integers for each variable.

[1] Continuous variables can take on any value on the real line, possibly between lower and upper bounds.

[2] Discrete variables can take on only integer values.

We will consider integer programming problems of the form

$$\mathbf{P}_d \qquad \min_{x \in \mathbf{X}_d} f(x). \tag{4.4}$$

4.1.3 Problems with Continuous and Discrete Variables

Next, we will allow for continuous and discrete independent variables.

We will use the notation

$$\mathbf{X} \triangleq \mathbf{X}_c \times \mathbf{X}_d, \tag{4.5a}$$
$$\mathbf{X}_c \triangleq \left\{ x \in \mathbb{R}^{n_c} \mid l^i \le x^i \le u^i,\ i \in \{1, \ldots, n_c\} \right\}, \tag{4.5b}$$

where the bounds on the continuous independent variables satisfy $-\infty \le l^i < u^i \le \infty$ for $i \in \{1, \ldots, n_c\}$, and the constraint set $\mathbf{X}_d \subset \mathbb{Z}^{n_d}$ for the discrete variables is a user-specified set with a finite, non-zero number of integers for each variable.

We will consider mixed-integer programming problems of the form

$$\mathbf{P}_{cd} \qquad \min_{x \in \mathbf{X}} f(x), \tag{4.6a}$$
$$\tag{4.6b}$$

where $x \triangleq (x_c, x_d) \in \mathbb{R}^{n_c} \times \mathbb{Z}^{n_d}$, $f \colon \mathbb{R}^{n_c} \times \mathbb{Z}^{n_d} \to \mathbb{R}$ and $\mathbf{X}$ is as in (4.5).

Now, we add inequality constraints on the dependent variables to (4.6) and obtain

$$\mathbf{P}_{cdg} \qquad \min_{x \in \mathbf{X}} f(x), \tag{4.7a}$$
$$g(x) \le 0, \tag{4.7b}$$

where everything is as in (4.6) and in addition $g \colon \mathbb{R}^{n_c} \times \mathbb{R}^{n_d} \to \mathbb{R}^m$ (for some $m \in \mathbb{N}$). We will assume that there exists an $x^* \in \mathbf{X}$ that satisfies $g(x^*) < 0$.

4.1.4 Problems whose Cost Function is Evaluated by a Building Simulation Program

Next, we will discuss problem $\mathbf{P}_c$ defined in (4.2) for the situation where the cost function $f \colon \mathbb{R}^n \to \mathbb{R}$ cannot be evaluated, but can be approximated numerically by approximating cost functions $f^* \colon \mathbb{R}^n \times \mathbb{R}^p_+ \to \mathbb{R}$, where the second argument is the precision parameter of the numerical solvers. This is typically the case when the cost is computed by a thermal building simulation program, such as EnergyPlus [CLW+01], TRNSYS [KDB76], or DOE-2 [WBB+93]. In such programs, computing the cost involves solving a system of partial and ordinary differential equations that is coupled to algebraic equations. In general, one cannot obtain an exact solution, but one can obtain an approximate numerical solution. Hence, the cost function $f(x)$ can only be approximated by an approximating cost function $f^*(x; \epsilon)$, where $\epsilon \in \mathbb{R}^q_+$ is a vector that contains

precision parameters of the numerical solvers. Consequently, the optimization algorithm can only be applied to $f^*(x;\epsilon)$ and not to $f(x)$.

In such thermal building simulation programs it is common that the termination criteria of the solvers that are used to solve the partial differential equations, ordinary differential equations, and algebraic equations depend on the independent variable x. Therefore, a perturbation of x can cause a change in the sequence of solver iterations, which causes the approximating cost functions $f^*(x;\epsilon)$ to be discontinuous in x. Furthermore, if variable step size integration methods are used, then the integration mesh can change from one simulation to the next. Therefore, part of the change in function values between different points is caused by a change of the number of solver iterations, and by a change of the integration mesh. Consequently, $f^*(\cdot;\epsilon)$ is discontinuous, and a descent direction for $f^*(\cdot;\epsilon)$ may not be a descent direction for $f(\cdot)$. Therefore, optimization algorithms can terminate at points that are non-optimal.

The best one can do in trying to solve optimization problems where the cost and constraint functions are evaluated by a thermal building simulation program that does not allow controlling the approximation error is to find points that are close to a local minimizer of $f(\cdot)$. Numerical experiments show that by using tight enough precision and starting the optimization algorithm with coarse initial values, one often comes close to a minimizer of $f(\cdot)$. Furthermore, by selecting different initial iterates for the optimization, or by using different optimization algorithms, one can increase the chance of finding a point that is close to a minimizer of $f(\cdot)$. However, even if the optimization terminates at a point that is non-optimal for $f(\cdot)$, one may have obtained a better system performance compared to not doing any optimization.

4.2 Algorithm Selection

In this section, we will discuss which of GenOpt's algorithms can be selected for the optimization problems that we introduced in Section 4.1.

4.2.1 Problem $\mathbf{P}_c$ with $n > 1$

To solve $\mathbf{P}_c$ with $n > 1$, the hybrid algorithm (Section 5.5, page 46) or the GPS implementation of the Hooke-Jeeves algorithm (Section 5.2.2, page 29) can be used. If $f(\cdot)$ is once continuously differentiable and has bounded level sets (or if the constraint set $\mathbf{X}$ defined in (4.1) is compact) then these algorithms construct accumulation points that are feasible stationary points of problem (4.2) (see Theorem 5.1.24).

Alternatively, the Discrete Armijo Gradient algorithm (Section 5.3, page 32) can be used. Every accumulation point of the Discrete Armijo Gradient algorithm is a feasible stationary point.

If $f(\cdot)$ is not continuously differentiable, or if $f(\cdot)$ must be approximated by an approximating cost function $f^*(\cdot;\epsilon)$ where the approximation error cannot be controlled, as described in Section 4.1.4, then $\mathbf{P}_c$ can only be solved heuristically. We recommend using the hybrid algorithm (Section 5.5, page 46), the GPS implementation of the Hooke-Jeeves algorithm (Section 5.2.2, page 29), a Particle Swarm Optimization algorithm (Section 5.4, page 36), or the Nelder-Mead's Simplex algorithm (Section 5.7, page 54).

The following approach reduces the risk of only finding a point which is non-optimal and far from a minimizer of $f(\cdot)$.

1. Selecting large values for the parameter `Step` in the optimization command file (see page 95),
2. selecting different initial iterates,
3. using the hybrid algorithm of Section 5.5, the GPS implementation of the Hooke-Jeeves algorithm, a Particle Swarm Optimization algorithm, and/or Nelder-Mead's Simplex algorithm and select the best of the solutions, and/or
4. doing a parametric study around the solution that has been obtained by the hybrid algorithm of Section 5.5, the GPS implementation of the Hooke-Jeeves algorithm, a Particle Swarm Optimization algorithm, and/or Nelder-Mead's Simplex algorithm. The parametric study can be done using the algorithms `Parametric` (Section 7.1, page 71) and/or `EquMesh` (Section 7.2, page 72).

If $f(\cdot)$ is continuously differentiable but must be approximated by approximating cost functions $f^*(\cdot;\epsilon)$ where the approximation error can be controlled as described in Section 4.1.4, then $\mathbf{P}_c$ can be solved using the hybrid algorithm (Section 5.5, page 46) or the GPS implementation of the Hooke-Jeeves algorithm (Section 5.2.2, page 29), both with the error control scheme described in the Model GPS algorithm 5.1.14 (page 22). The error control scheme can be implemented using the value of GenOpt's variable `stepNumber` (page 77) and GenOpt's pre-processing capabilities (Section 11.2, page 98). A more detailed description of how to use the error control scheme can be found in [PW03, WP03].

4.2.2 Problem $\mathbf{P}_{cg}$ with $n > 1$

To solve $\mathbf{P}_{cg}$, the hybrid algorithm (Section 5.5, page 46) or the GPS implementation of the Hooke-Jeeves algorithm (Section 5.2.2, page 29) can be used. Constraints $g(\cdot) \leq 0$ can be implemented using barrier and penalty functions (Section 8, page 74).

If $f(\cdot)$ or $g(\cdot)$ are not continuously differentiable, we recommend using the hybrid algorithm (Section 5.5, page 46) or the GPS implementation of the Hooke-Jeeves algorithm (Section 5.2.2, page 29), and implement the constraints $g(\cdot) \leq 0$ using barrier and penalty functions (Section 8, page 74). To reduce

the risk of terminating far from a minimum point of $f(\cdot)$, we recommend the same measures as for solving $\mathbf{P}_c$.

4.2.3 Problem $\mathbf{P}_c$ with $n = 1$

To solve $\mathbf{P}_c$ with $n = 1$, any of the interval division algorithms can be used (Section 6.1, page 65). Since only few function evaluations are required for parametric studies, the algorithm `Parametric` can also be used for this problem (Section 7.1, page 71). We recommend to do a parametric study if $f(\cdot)$ is expected to have several local minima.

4.2.4 Problem $\mathbf{P}_{cg}$ with $n = 1$

To solve $\mathbf{P}_{cg}$ with $n = 1$, the same applies as for $\mathbf{P}_c$ with $n = 1$. Constraints $g(\cdot) \leq 0$ can be implemented by setting the penalty weighting factor μ in (8.8) to a large value. This may still cause small constraint violations, but it is easy to check whether the violation is acceptable.

4.2.5 Problem $\mathbf{P}_d$

To solve $\mathbf{P}_d$, a Particle Swarm Optimization algorithm can be used (Section 5.4, page 36).

4.2.6 Problem $\mathbf{P}_{cd}$ and $\mathbf{P}_{cdg}$

To solve $\mathbf{P}_{cd}$, or $\mathbf{P}_{cdg}$, the hybrid algorithm (Section 5.5, page 46) or a Particle Swarm Optimization algorithm can be used (Section 5.4, page 36).

4.2.7 Functions with Several Local Minima

If the problem has several local minima, we recommend using the hybrid algorithm (Section 5.5, page 46), a Particle Swarm Optimization algorithm (Section 5.4, page 36), or run any of the other algorithms multiple times with different initial values.

5 Algorithms for Multi-Dimensional Optimization

5.1 Generalized Pattern Search Methods (Analysis)

Generalized Pattern Search (GPS) algorithms are derivative free optimization algorithms for the minimization of problem $\mathbf{P}_c$ and $\mathbf{P}_{cg}$, defined in (4.2) and (4.3), respectively. We will present the GPS algorithms for the case where the function $f(\cdot)$ cannot be evaluated exactly, but can be approximated by functions $f^*\colon \mathbb{R}^n \times \mathbb{R}^q_+ \to \mathbb{R}$, where the second argument $\epsilon \in \mathbb{R}^q_+$ is the precision parameter of the PDE, ODE, and algebraic equation solvers. Under the assumption that the cost function is continuously differentiable, all the accumulation points constructed by the GPS algorithms are stationary, while under the assumption that $f(\cdot)$ is only locally Lipschitz continuous, the GPS algorithms converge to points at which the Clarke generalized directional derivatives are nonnegative in predefined directions.

Obviously, the explanations are similar for problems where $f(\cdot)$ can be evaluated exactly, except that the scheme to control ϵ is not applicable, and the approximate functions $f^*(\cdot;\epsilon)$ are replaced by $f(\cdot)$.

What GPS algorithms have in common is that they define the construction of a mesh in $\mathbb{R}^n$, which is then explored according to some rule that differs between the various members of the family of GPS algorithms. If no decrease in cost is obtained on mesh points around the current iterate, then the mesh is refined and the process is repeated.

We now start with explaining the general framework of GPS algorithm that will be used to implement different instances of GPS algorithms in GenOpt. The discussion follows the more detailed description of [PW03].

5.1.1 Assumptions

We will assume that $f(\cdot)$ and its approximating functions $\{f(\cdot;\epsilon)\}_{\epsilon\in\mathbb{R}^q_+}$ have the following properties.

Assumption 5.1.1

1. *There exists an error bound function* $\varphi\colon \mathbb{R}^q_+ \to \mathbb{R}_+$ *such that for any bounded set* $\mathbf{S}\subset\mathbf{X}$*, there exists an* $\epsilon_{\mathbf{S}}\in\mathbb{R}^q_+$ *and a scalar* $K_{\mathbf{S}}\in(0,\infty)$ *such that for all* $x\in\mathbf{S}$ *and for all* $\epsilon\in\mathbb{R}^q_+$*, with* $\epsilon\le\epsilon_{\mathbf{S}}$ [1]*,*

$$|\, f^*(x;\epsilon) - f(x)| \le K_{\mathbf{S}}\,\varphi(\epsilon). \tag{5.1}$$

Furthermore,

$$\lim_{\epsilon\to 0}\varphi(\epsilon) = 0. \tag{5.2}$$

2. *The function* $f\colon\mathbb{R}^n\to\mathbb{R}$ *is at least locally Lipschitz continuous.* □

Remark 5.1.2 The functions $\{f^*(\cdot;\epsilon)\}_{\epsilon\in\mathbb{R}^q_+}$ may be discontinuous. □

Next, we state an assumption on the level sets of the family of approximate functions. To do so, we first define the notion of a level set.

Definition 5.1.3 (Level Set) *Given a function* $f\colon\mathbb{R}^n\to\mathbb{R}$ *and an* $\alpha\in\mathbb{R}$*, such that* $\alpha\ge\inf_{x\in\mathbb{R}^n} f(x)$*, we will say that the set* $\mathbf{L}_\alpha(f)\subset\mathbb{R}^n$*, defined as*

$$\mathbf{L}_\alpha(f) \triangleq \{x\in\mathbb{R}^n \mid f(x)\le\alpha\}, \tag{5.3}$$

is a level set *of* $f(\cdot)$*, parametrized by* α*.* □

Assumption 5.1.4 (Compactness of Level Sets) *Let* $\{f^*(\cdot;\epsilon)\}_{\epsilon\in\mathbb{R}^q_+}$ *be as in Assumption 5.1.1 and let* $\mathbf{X}\subset\mathbb{R}^n$ *be the constraint set. Let* $x_0\in\mathbf{X}$ *be the initial iterate and* $\epsilon_0\in\mathbb{R}^q_+$ *be the initial precision setting of the numerical solvers. Then, we assume that there exists a compact set* $\mathbf{C}\subset\mathbb{R}^n$ *such that*

$$\mathbf{L}_{f^*(x_0;\epsilon_0)}(f^*(\cdot;\epsilon))\cap\mathbf{X}\subset\mathbf{C}, \qquad \forall\,\epsilon\le\epsilon_0. \tag{5.4}$$

□

5.1.2 Geometric Aspects of the Algorithms

A major aspect of any GPS algorithm is the rule for generating the meshes on which the searches are conducted. The main difference between our rule for mesh generation and those of others, such as the one of Audet's and Dennis [AD03], is that we use a different rule for mesh refinement, which results in our meshes being nested, and hence simplifies the explanation of the geometry of mesh generation. As far as we can tell, our simplification has no impact on

[1] For $\epsilon\in\mathbb{R}^q_+$, by $\epsilon\le\epsilon_{\mathbf{S}}$, we mean that $\epsilon^i\le\epsilon^i_{\mathbf{S}}$, for all $i\in\{1,\dots,q\}$.

computational efficiency.

The k-th iteration of our GPS algorithms has the following structure. We begin with the current iterate x_k, with the precision settings of the PDE, ODE, and algebraic equation solvers ϵ, and with the mesh $\mathbb{M}_k$. A set-valued map is used to select a finite subset of mesh points in $\mathbb{M}_k$, for the so-called "global search". If this set contains a point x' such that $f^*(x';\epsilon) < f^*(x_k;\epsilon)$, then we set $x_{k+1} = x'$, $\mathbb{M}_{k+1} = \mathbb{M}_k$, and update the index k to $k+1$. If the global search set does not yield a point of lower cost, we proceed to a "local search", which consists of evaluating $f^*(\cdot;\epsilon)$ on a set of neighbors of x_k in the mesh $\mathbb{M}_k$. If a point x' of lower cost is found, then we set $x_{k+1} = x'$, $\mathbb{M}_{k+1} = \mathbb{M}_k$, and update the index k to $k+1$. If the local search also fails to produce an improvement, then the mesh $\mathbb{M}_k$ is subdivided to yield a finer mesh $\mathbb{M}_{k+1}$, and the precision of the PDE, ODE, and algebraic equation solvers, ϵ, is increased according to a prescribed rule. After updating k to $k+1$, the entire process is repeated.

We will now flesh out the geometric details of our GPS algorithms. We begin with the construction of the meshes.

a) Generation of the Meshes

Before we can explain how the mesh is to be generated, we must introduce the notions of a positive combination and of a positive span, as defined by Davis [Dav54], and that of a base direction matrix.

Definition 5.1.5 (Positive Combination, Positive Span)

1. *A* positive combination *of vectors* $\{v_i\}_{i=1}^p$ *is a linear combination* $\sum_{i=1}^p \lambda_i v_i$ *with* $\lambda_i \geq 0$ *for all* $i \in \{1, \ldots, p\}$.
2. *A* positive span *for a subspace* $\mathbf{S} \subset \mathbb{R}^n$ *is a set of vectors* $\{v_i\}_{i=1}^p$ *such that every* $x \in \mathbf{S}$ *can be expressed as a positive combination of the vectors* $\{v_i\}_{i=1}^p$. *The matrix defined by* $V \triangleq [v_1, \ldots, v_p]$ *is said to be a* positive spanning matrix.
3. *Let the subspace* $\mathbf{S} \subset \mathbb{R}^n$ *be of dimension* m *and* $V \in \mathbb{R}^{n \times p}$ *be a positive spanning matrix for* $\mathbf{S}$. *If* $p = m+1$, *then* V *is said to be a* minimal positive spanning matrix. □

In [Dav54, CP00], a positive basis for a subspace $\mathbf{S} \subset \mathbb{R}^n$ is defined as a set of positively independent vectors whose positive span is $\mathbf{S}$. Note that a positive basis is different from a minimal positive spanning set. For example, if $\mathbf{S} = \mathbb{R}^2$, the set $\{e_1, e_2, -e_1, -e_2\}$ is a positive basis but not a minimal positive spanning set. A minimal positive spanning set is $\{e_1, e_2, -(e_1+e_2)\}$.

We will denote by $\mathbb{S}$ the set of all matrices whose columns positively span $\mathbb{R}^n$.

Next, we define a *base direction matrix*. We will use the columns of the base direction matrix to specify the mesh and hence the search directions. The base direction matrix will be fixed for all iterations.

Definition 5.1.6 (Base Direction Matrix) *Let $\mathbb{S}$ be the set of all matrices whose columns positively span $\mathbb{R}^n$. Then, the* base direction matrix $\widehat{D}$ *is any matrix satisfying*

$$\widehat{D} \in \mathbb{Q}^{n\times p} \cap \mathbb{S} \tag{5.5}$$

where $p > n$ is any arbitrary but finite natural number. □

Remark 5.1.7 The fact that the matrix $\widehat{D}$ has only rational elements makes it very easy to establish the minimal distance between distinct mesh points. At the same time, from a computational point of view, requiring $\widehat{D} \in \mathbb{Q}^{n\times p} \cap \mathbb{S}$ rather than $\widehat{D} \in \mathbb{R}^{n\times p} \cap \mathbb{S}$ does not result in any practical inconvenience. □

Note that the base direction matrix $\widehat{D}$ may not be a minimal positive spanning matrix, e.g., for the one-dimensional case, $\widehat{D} = [-1,\ 1,\ 1.1]$ would not be minimal. Hence, $\widehat{D}$ can be used to generate a set $\mathbb{D}_{\widehat{D}}$, which we define as the set of all submatrices of $\widehat{D}$ (constructed by deleting columns of $\widehat{D}$) whose column vectors positively span $\mathbb{R}^n$.

The meshes, over which our algorithms search, are defined iteratively, as follows.

Definition 5.1.8 (k-th Mesh) *Let $x_0 \in \mathbf{X}$, $r, s_0, k \in \mathbb{N}$, with $r > 1$, $\{t_i\}_{i=0}^{k-1} \subset \mathbb{N}$, and the base direction matrix $\widehat{D} \in \mathbb{Q}^{n\times p} \cap \mathbb{S}$ be given, and let*

$$\Delta_k \triangleq \frac{1}{r^{s_k}}, \tag{5.6}$$

where for $k > 0$

$$s_k = s_0 + \sum_{i=0}^{k-1} t_i. \tag{5.7}$$

Then we define the mesh $\mathbb{M}_k$ *by*

$$\mathbb{M}_k \triangleq \{x_0 + \Delta_k \widehat{D}\, m \mid m \in \mathbb{N}^p\}. \tag{5.8}$$

□

It should be clear from the definition of the meshes that whenever $t_k > 0$, the mesh $\mathbb{M}_{k+1}$ is obtained from the mesh $\mathbb{M}_k$ by dividing the intervals between neighboring points of the mesh $\mathbb{M}_k$ into r^{t_k} subintervals by adding additional mesh points. Therefore, it is clear that the meshes are nested, i.e., $\mathbb{M}_k \subset \mathbb{M}_{k+1}$ with equality if $\Delta_{k+1} = \Delta_k$.

We now present two examples: first a simple example of a mesh that is generated by a minimal positive spanning matrix, and then an example of a mesh generation using a more complicated base direction matrix $\widehat{D}$.

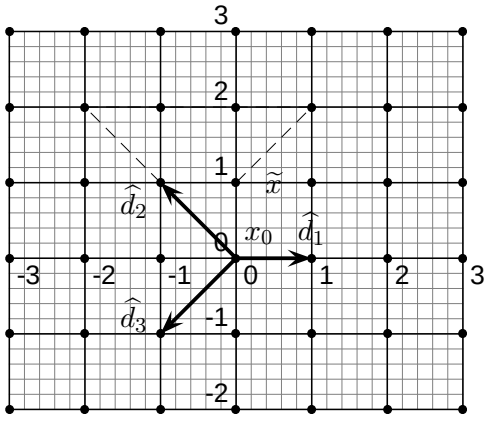


Figure 5.1: *Minimal positive spanning matrix* $\widehat{D} = [\widehat{d}_1, \widehat{d}_2, \widehat{d}_3]$ *and generated mesh in* $\mathbb{R}^2$.

Example 5.1.9 In Fig. 5.1, the base direction matrix $\widehat{D}$ is a minimal positive spanning matrix, defined by

$$\widehat{D} = \begin{pmatrix} \widehat{d}_1 & \widehat{d}_2 & \widehat{d}_3 \end{pmatrix} \triangleq \begin{pmatrix} 1 & -1 & -1 \\ 0 & 1 & -1 \end{pmatrix}. \tag{5.9}$$

In Fig. 1, the bullets ($\bullet$) are the mesh points of the mesh $\mathbb{M}_k = \{0 + 1\,\widehat{D}\,m \mid m \in \mathbb{N}^3\}$. For example, in Fig. 1, $\widetilde{x} = \widehat{D}\,m$, with $m = (3,\ 2,\ 1)^T$. □

Next we present a mesh constructed using a more complicated base direction matrix.

Example 5.1.10 Fig. 5.2 shows a mesh generated using $x_0 = 0$, $\Delta_k = 1$ and the base direction matrix

$$\widehat{D} = \begin{pmatrix} \widehat{d}_1 & \widehat{d}_2 & \widehat{d}_3 \end{pmatrix} = \begin{pmatrix} 1 & -0.5 & -0.75 \\ 0 & 1 & -0.75 \end{pmatrix}. \tag{5.10}$$

Fig. 5.2(a) shows the vectors $\{\widehat{d}_i\}_{i=1}^3$ (bold arrows) and all possible mesh points of the form $\widehat{D}\,v$ with $v = (n,\ 0,\ 0)^T$, $v = (0,\ n,\ 0)^T$, and $v = (0,\ 0,\ n)^T$ where $n \in \mathbb{N}$. Each arrow points to a mesh point and indicates how the base vectors $\{\widehat{d}_i\}_{i=1}^3$ are added to obtain the mesh points. Fig. 5.2(b) shows the set of all mesh points of the form $\widehat{D}\,v$ with $v = (n,\ m,\ 0)^T$ and $v = (n,\ 0,\ m)^T$ where $n, m \in \mathbb{N}$. For example, the point labeled with $\widetilde{x}$ is given by $\widetilde{x} = \widehat{D}\,v$ where $v = (2,\ 1,\ 0)^T$. In Fig. 5.2(c), more mesh points are drawn by adding some positive multiple of $\widehat{d}_2$ to some mesh points that have been generated in Fig. 5.2(b). For clarity, not all possible mesh points are drawn. In Fig. 5.2(d), additional mesh points are generated by adding some positive multiple of $\widehat{d}_3$ to some mesh points of Fig. 5.2(c). Fig. 5.2(e) finally contains all possible mesh points, now indicated by bullets ($\bullet$). For clarity, only the vectors $\{\widehat{d}_i\}_{i=1}^3$ are drawn in Fig. 5.2(e). □

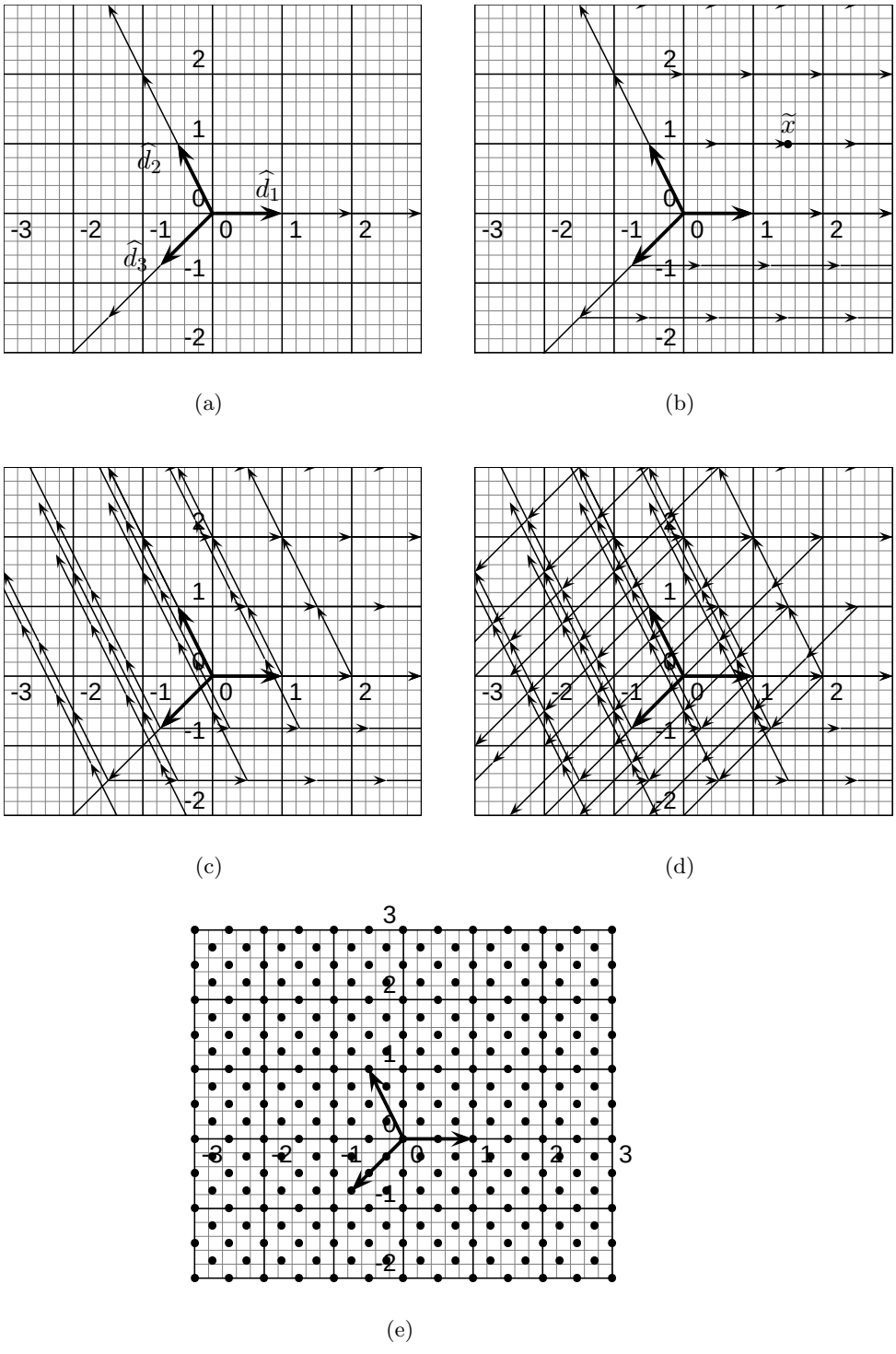


Figure 5.2: Generation of a mesh in $\mathbb{R}^2$.

b) Global and Local Search Set

We will now characterize the set-valued maps that determine the mesh points for the "global" and "local" searches. Note that the images of these maps may depend on the entire history of the computation.

Definition 5.1.11 (Search Direction Matrices) *Let $\mathbb{S}$ be the set of all matrices whose column vectors positively span $\mathbb{R}^n$. Given a base direction matrix $\widehat{D}$, we define the set of* search direction matrices *to be*

$$\mathbb{D}_{\widehat{D}} \triangleq \{D \mid D \subset \widehat{D} \cap \mathbb{S}\} \tag{5.11}$$

where the matrix D is constructed by deleting columns of $\widehat{D}$. □

Definition 5.1.12 *Let $\underline{\mathbf{X}}_k \subset \mathbb{R}^n$ and $\underline{\mathbf{\Delta}}_k \subset \mathbb{Q}_+$ be the sets of all sequences containing k elements, let $\mathbb{M}_k$ be the current mesh, let $\mathbb{D}_{\widehat{D}}$ be the set of search direction matrices, and let $\epsilon \in \mathbb{R}^q_+$ be the precision of the numerical solvers.*

1. *We define the* global search set map *to be any set-valued map*

$$\gamma_k \colon \underline{\mathbf{X}}_k \times \underline{\mathbf{\Delta}}_k \times \mathbb{R}^q_+ \to \left(2^{\mathbb{M}_k} \cap \mathbf{X}\right) \cup \emptyset \tag{5.12a}$$

   whose image $\gamma_k(\underline{x}_k, \underline{\Delta}_k, \epsilon)$ contains only a finite number of mesh points.

2. *We define the* local search direction map *to be any map*

$$\delta_{\widehat{D},k} \colon \underline{\mathbf{X}}_k \times \underline{\mathbf{\Delta}}_k \to \mathbb{D}_{\widehat{D}}. \tag{5.12b}$$

3. *We will call $\mathcal{G}_k \triangleq \gamma_k(\underline{x}_k, \underline{\Delta}_k, \epsilon)$ the* global search set.

4. *With $D_k = \delta_{\widehat{D},k}(\underline{x}_k, \underline{\Delta}_k)$, we will call*

$$\mathcal{L}_k \triangleq \{x_k + \Delta_k D_k e_j \mid j = 1, \ldots, \mathrm{card}(D_k)\} \cap \mathbf{X} \tag{5.12c}$$

   the local search set.

□

Remark 5.1.13

1. The map $\gamma_k(\cdot,\cdot,\cdot)$ can be dynamic in the sense that if $\{x_{k_i}\}_{i=0}^{I} \triangleq \gamma_k(\underline{x}_k, \underline{\Delta}_k, \epsilon)$, then the rule for selecting $x_{k_{\widehat{i}}}$, $1 \le \widehat{i} \le I$, can depend on $\{x_{k_i}\}_{i=0}^{\widehat{i}-1}$ and $\{f^*(x_{k_i};\epsilon)\}_{i=0}^{\widehat{i}-1}$. It is only important that the global search terminates after a finite number of computations, and that $\mathcal{G}_k \subset (2^{\mathbb{M}_k} \cap \mathbf{X}) \cup \emptyset$.
2. As we shall see, the global search affects only the efficiency of the algorithm but not its convergence properties. Any heuristic procedure that leads to a finite number of function evaluations can be used for $\gamma_k(\cdot,\cdot,\cdot)$.
3. The empty set is included in the range of $\gamma_k(\cdot,\cdot,\cdot)$ to allow omitting the global search.
4. Since the range of $\delta_{\widehat{D},k}(\cdot,\cdot)$ is $\mathbb{D}_{\widehat{D}}$, any image of $\delta_{\widehat{D},k}(\cdot,\cdot)$ is a positive spanning matrix.

□

5.1.3 A Model Adaptive Precision GPS Algorithm

We are now ready to present a model generalized pattern search algorithm with adaptive precision function evaluations.

Algorithm 5.1.14 (Model GPS Algorithm)

Data:	Initial iterate $x_0 \in \mathbf{X}$. $r \in \mathbb{N}$, $r > 1$, used to compute Δ_k. Initial mesh size exponent $s_0 \in \mathbb{N}$. Base direction matrix $\widehat{D} \in \mathbb{Q}^{n\times p} \cap \mathbb{S}$ (see Definition 5.1.6).
Maps:	Global search set map $\gamma_k \colon \underline{\mathbf{X}_k} \times \underline{\mathbf{\Delta}}_k \times \mathbb{R}_+^q \to \left(2^{\mathbb{M}_k} \cap \mathbf{X}\right) \cup \emptyset$. Local search direction map $\delta_{\widehat{D},k} \colon \underline{\mathbf{X}_k} \times \underline{\mathbf{\Delta}}_k \to \mathbb{D}_{\widehat{D}}$ (see Definition 5.1.12). Function $\rho \colon \mathbb{R}_+ \to \mathbb{R}_+^q$ (to assign ϵ), such that the composition $\varphi \circ \rho \colon \mathbb{R}_+ \to \mathbb{R}_+$ is strictly monotone decreasing and satisfies $\varphi(\rho(\Delta))/\Delta \to 0$, as $\Delta \to 0$.
Step 0:	Initialize $k = 0$, $\Delta_0 = 1/r^{s_0}$, and $\epsilon = \rho(1)$.
Step 1:	Global Search Construct the global search set $\mathcal{G}_k = \gamma_k(\underline{x}_k, \underline{\Delta}_k, \epsilon)$. If $f^*(x';\epsilon) < f^*(x_k;\epsilon)$ for any $x' \in \mathcal{G}_k$, go to Step 3. Else, go to Step 2.
Step 2:	Local Search Construct the search direction matrix $D_k = \delta_{\widehat{D},k}(\underline{x}_k, \underline{\Delta}_k)$. Construct $\mathcal{L}_k \triangleq \{x_k + \Delta_k\, D_k\, e_j \mid j = 1, \ldots, \mathrm{card}(D_k)\} \cap \mathbf{X}$ and evaluate $f^*(\cdot;\epsilon)$ for any $x' \in \mathcal{L}_k$ until some $x' \in \mathcal{L}_k$ satisfying $f^*(x';\epsilon) < f^*(x_k;\epsilon)$ is obtained, or until all points in $\mathcal{L}_k$ are evaluated.
Step 3:	Parameter Update If there exists an $x' \in \mathcal{G}_k \cup \mathcal{L}_k$ satisfying $f^*(x';\epsilon) < f^*(x_k;\epsilon)$, set $x_{k+1} = x'$, $s_{k+1} = s_k$, $\Delta_{k+1} = \Delta_k$, and do not change ϵ. Else, set $x_{k+1} = x_k$, $s_{k+1} = s_k + t_k$, with $t_k \in \mathbb{N}_+$ arbitrary, $\Delta_{k+1} = 1/r^{s_{k+1}}$, $\epsilon = \rho(\Delta_{k+1}/\Delta_0)$.
Step 4:	Replace k by $k + 1$, and go to Step 1.

Remark 5.1.15

1. If the optimization is started with $\epsilon = \rho(1)$ too small, the computation time may become unnecessarily large. Therefore, in implementing the Model GPS Algorithm, one may allow to redefine the function $\rho(\cdot)$ by $\rho(\cdot) \leftarrow c\,\rho(\cdot)$, with $c \in (0, 1)$, to decrease the initial number of solver iterations. Redefining the function $\rho(\cdot)$ is allowed over a preset number of GPS iterations.

2. To ensure that ϵ does not depend on the scaling of Δ_0, we normalized the argument of $\rho(\cdot)$. In particular, we want to decouple the precision settings of the solvers from the user's choice of the initial mesh divider.

3. Audet and Dennis [AD03] increase and decrease the mesh divider using the formula $\Delta_{k+1} = \tau^m\, \Delta_k$ where $\tau \in \mathbb{Q}$, $\tau > 1$, and m is any element of

$\mathbb{Z}$. Thus, our mesh construction is a special case of Audet's and Dennis' construction since we set $\tau = 1/r$, with $r \in \mathbb{N}_+$, $r \geq 2$ (so that, $\tau < 1$) and $m \in \mathbb{N}$. We prefer our construction because it leads to a simpler geometric explanation. In the Appendix, we present a modified version of the algorithm of Audet and Dennis, and show that our analysis remains valid.

4. In Step 2, once a decrease of the cost function is obtained, one can proceed to Step 3. However, one is allowed to evaluate the approximating cost function at more points in $\mathcal{L}_k$ in an attempt to obtain a bigger reduction in cost. However, one is allowed to proceed to Step 3 only after either a cost decrease has been found, or after *all* points in $\mathcal{L}_k$ are tested.

5. In Step 3, we are not restricted to accepting the $x' \in \mathcal{G}_k \cup \mathcal{L}_k$ that gives lowest cost value. But the mesh divider Δ_k is reduced *only* if there exists no $x' \in \mathcal{G}_k \cup \mathcal{L}_k$ satisfying $f^*(x'; \epsilon) < f^*(x_k; \epsilon)$. □

5.1.4 Convergence Results

a) Unconstrained Minimization

We will now present the convergence results for the GPS algorithms. See [PW03] for a detailed discussion and convergence proofs.

We will now present the convergence properties of the Model GPS Algorithm 5.1.14 on unconstrained minimization problems, i.e., for $\mathbf{X} = \mathbb{R}^n$.

First, we will need the notion of a *refining subsequence*, which we define as follows:

Definition 5.1.16 (Refining Subsequence) *Consider a sequence $\{x_k\}_{k=0}^{\infty}$ constructed by Model GPS Algorithm 5.1.14. We will say that the subsequence $\{x_k\}_{k\in\mathbf{K}}$ is the* refining subsequence*, if $\Delta_{k+1} < \Delta_k$ for all $k \in \mathbf{K}$, and $\Delta_{k+1} = \Delta_k$ for all $k \notin \mathbf{K}$.* □

To state the convergence result for the case where $f(\cdot)$ is only Lipschitz continuous, we recall the definition of Clarke's generalized directional derivative [Cla90]:

Definition 5.1.17 (Clarke's Generalized Directional Derivative) *Let $f\colon \mathbb{R}^n \to \mathbb{R}$ be locally Lipschitz continuous at the point $x^* \in \mathbb{R}^n$. Then,* Clarke's generalized directional derivative *of $f(\cdot)$ at x^* in the direction $h \in \mathbb{R}^n$ is defined by*

$$\mathrm{d}^0 f(x^*;\, h) \triangleq \limsup_{\substack{x \to x^* \\ t \downarrow 0}} \frac{f(x + t\,h) - f(x)}{t}. \tag{5.13}$$

□

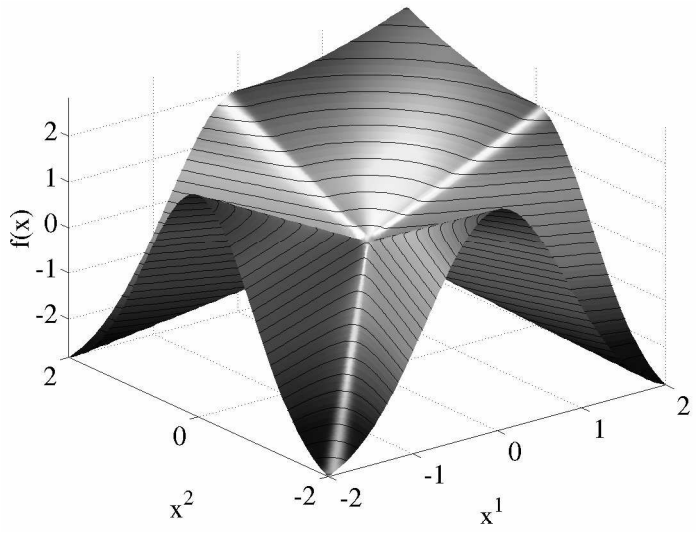


Figure 5.3: *Visualization of equation* (5.15).

Theorem 5.1.18 *Suppose that Assumptions 5.1.1 and 5.1.4 are satisfied and let $x^* \in \mathbb{R}^n$ be an accumulation point of a refining subsequence $\{x_k\}_{k\in\mathbf{K}}$, constructed by Model GPS Algorithm 5.1.14. Let d be any column of the base direction matrix $\widehat{D}$ along which $f^*(\cdot;\cdot)$ was evaluated for infinitely many iterates in the subsequence $\{x_k\}_{k\in\mathbf{K}}$. Then,*

$$\mathrm{d}^0 f(x^*;\, d) \geq 0. \tag{5.14}$$

□

Remark 5.1.19 Note that (5.14) is not a standard optimality condition since it holds only for certain directions d. Consider, for example, the Lipschitz continuous function

$$f(x) \triangleq \begin{cases} \|x\|, & \text{if } x^1 > 0 \text{ and } x^2 > 0, \\ \|x\| \cos\bigl(4 \arccos\bigl(x^1/\|x\|\bigr)\bigr), & \text{otherwise,} \end{cases} \tag{5.15}$$

which is shown in Fig. 5.3. This function is not differentiable at the origin, but it does have directional derivatives everywhere. At the origin $x^* = 0$, we have $\mathrm{d}f(x^*;d) = 1$ for $d \in \{\pm e_1,\ \pm e_2\}$, but the directional derivative along $s = (-1,\ -1)^T$ is $\mathrm{d}f(x^*;s) = -\sqrt{2}$.
Using the Hooke-Jeeves algorithm with initial value $x_0 = (-1,\, 0)^T$ and $\Delta = \Delta_0 = 1$, we would converge to the origin, a point that possess some negative directional derivatives. □

We now state that pattern search algorithms with adaptive precision function evaluations converge to stationary points.

Theorem 5.1.20 (Convergence to a Stationary Point) *Suppose that Assumptions 5.1.1 and 5.1.4 are satisfied and, in addition, that $f(\cdot)$ is once continuously differentiable and that $\mathbf{X} = \mathbb{R}^n$. Let $x^* \in \mathbb{R}^n$ be an accumulation point of a refining subsequence $\{x_k\}_{k\in\mathbf{K}}$, constructed by Model GPS Algorithm 5.1.14. Then,*

$$\nabla f(x^*) = 0. \tag{5.16}$$

□

b) Box-Constrained Minimization

We now extend our convergence proofs to the box-constrained problem (4.2), by following the arguments in Audet and Dennis [AD03]. The more general case of linearly-constrained problems is discussed in [AD03, PW03].

First, we introduce the notion of a tangent cone and a normal cone, which are defined as follows:

Definition 5.1.21 (Tangent and Normal Cone)

1. *Let $\mathbf{X} \subset \mathbb{R}^n$. Then, we define the* tangent cone *to $\mathbf{X}$ at a point $x^* \in \mathbf{X}$ by*

$$\mathbf{T}_{\mathbf{X}}(x^*) \triangleq \overline{\{\mu\,(x - x^*) \mid \mu \geq 0,\, x \in \mathbf{X}\}}. \tag{5.17a}$$

2. *Let $\mathbf{T}_{\mathbf{X}}(x^*)$ be as above. Then, we define the* normal cone *to $\mathbf{X}$ at $x^* \in \mathbf{X}$ by*

$$\mathbf{N}_{\mathbf{X}}(x^*) \triangleq \{v \in \mathbb{R}^n \mid \forall\, t \in \mathbf{T}_{\mathbf{X}}(x^*),\, \langle v,\, t\rangle \leq 0\}. \tag{5.17b}$$

□

Next, we introduce the concept of conformity of a pattern to a constraint set (see [AD03]), which will enable us to extend the convergence results for our Model GPS Algorithm 5.1.14 from unconstrained optimization problems to box-constrained optimization problems.

Definition 5.1.22 *The function $\delta_{\widehat{D},k}\colon \underline{\mathbf{X}}_k \times \underline{\mathbf{\Delta}}_k \to \mathbb{D}_{\widehat{D}}$ is said to* conform *to the feasible set $\mathbf{X}$, if for some $\rho > 0$ and for each $x^* \in \partial\mathbf{X}$ satisfying $\|x^* - x_k\| < \rho$, the tangent cone $\mathbf{T}_{\mathbf{X}}(x^*)$ can be generated by nonnegative linear combinations of the columns of a subset $D_{x^*}(x_k) \subset D_k = \delta_{\widehat{D},k}(\underline{x}_k, \underline{\Delta}_k)$.*
Furthermore, we define $D_{x^*}(\cdot)$ to be such that all its columns belong to $\mathbf{T}_{\mathbf{X}}(x^*)$.

□

Remark 5.1.23 The definition that all columns of $D_{x^*}(\cdot)$ belong to $\mathbf{T}_{\mathbf{X}}(x^*)$ facilitates the extension of Theorem 5.1.18 to the constraint case. □

We can now state that the accumulation points generated by Model GPS Algorithm 5.1.14 are feasible stationary points of problem (4.2).

Theorem 5.1.24 (Convergence to a Feasible Stationary Point) *Suppose Assumptions 5.1.1 and 5.1.4 are satisfied and that $f(\cdot)$ is once continuously differentiable. Let $x^* \in \mathbf{X}$ be an accumulation point of a refining subsequence $\{x_k\}_{k\in\mathbf{K}}$ constructed by Model GPS Algorithm 5.1.14 in solving problem* (4.2).

If there exists a $k^* \in \mathbb{N}$ such that for all $k > k^*$, the local search direction maps $\delta_{\widehat{D},k}\colon \underline{\mathbf{X}}_k \times \underline{\mathbf{\Delta}}_k \to \mathbb{D}_{\widehat{D}}$ conform to the feasible set $\mathbf{X}$, then

$$\langle \nabla f(x^*),\, t\rangle \geq 0, \qquad \forall\, t \in \mathbf{T}_{\mathbf{X}}(x^*), \tag{5.18a}$$

and

$$-\nabla f(x^*) \in \mathbf{N}_{\mathbf{X}}(x^*). \tag{5.18b}$$

When the function $f(\cdot)$ is only locally Lipschitz continuous, we obtain following corollary which follows directly from Theorem 5.1.18 and equation (5.18a).

Corollary 5.1.25 *Suppose that the assumptions of Theorem 5.1.24 are satisfied, but $f(\cdot)$ were only locally Lipschitz continuous. Then,*

$$\mathrm{d}^0 f(x^*;\, d) \geq 0, \qquad \forall\, d \in D_{x^*}(x^*). \tag{5.19}$$

□

5.2 Generalized Pattern Search Methods (Implementations)

We will now present different implementations of the Generalized Pattern Search (GPS) algorithms. They all use the Model GPS Algorithm 5.1.14 to solve problem $\mathbf{P}_c$ defined in (4.2). The problem $\mathbf{P}_{cg}$ defined in (4.3) can be solved by using penalty functions as described in Section 8.2.

We will discuss the implementations for the case where the function $f(\cdot)$ cannot be evaluated exactly, but will be approximated by functions $f^*\colon \mathbb{R}^n \times \mathbb{R}^q_+ \to \mathbb{R}$, where the second argument $\epsilon \in \mathbb{R}^q_+$ is the precision parameter of the PDE, ODE, and algebraic equation solvers. This includes the case where ϵ is not varied during the optimization, in which case the explanations are identical, except that the scheme to control ϵ is not applicable, and the approximate functions $f^*(\cdot;\epsilon)$ are replaced by $f(\cdot)$.

If the cost function $f(\cdot)$ is approximated by functions $\{f^*(\cdot;\epsilon)\}_{\epsilon\in\mathbb{R}^q_+}$ with adaptive precision ϵ, then the function $\rho\colon \mathbb{R}_+ \to \mathbb{R}^q_+$ (to assign ϵ) can be implemented by using GenOpt's pre-processing capability (see Section 11.2).

5.2.1 Coordinate Search Algorithm

We will now present the implementation of the Coordinate Search algorithm with adaptive precision function evaluations using the Model GPS Algorithm 5.1.14. To simplify the implementation, we assign $f^*(x;\epsilon) = \infty$ for all $x \notin \mathbf{X}$ where $\mathbf{X}$ is defined in (4.1).

a) Algorithm Parameters

The base direction matrix is defined as

$$\widehat{D} \triangleq [+s^1\, e_1,\, -s^1\, e_1, \ldots,\, +s^n\, e_n,\, -s^n\, e_n] \tag{5.20}$$

where $s^i \in \mathbb{R}$, $i \in \{1, \ldots, n\}$, is a scaling for each parameter (specified by GenOpt's parameter `Step`).

$r \in \mathbb{N}$, $r > 1$, used to compute the mesh size divider is defined by the parameter `MeshSizeDivider`, the initial value for the mesh size exponent $s_0 \in \mathbb{N}$ is defined by the parameter `InitialMeshSizeExponent`, and the mesh size exponent increment t_k is fixed for all $k \in \mathbb{N}$ and defined by the parameter `MeshSizeExponentIncrement`.

b) Global Search

In the Coordinate Search Algorithm, there is no global search. Thus, $\mathcal{G}_k = \emptyset$ for all $k \in \mathbb{N}$.

c) Local Search

The local search direction map is given by $D_k = \delta_{\widehat{D},k}(\underline{x}_k, \underline{\Delta}_k) \triangleq \widehat{D}$ for all $k \in \mathbb{N}$.

The local search set $\mathcal{G}_k$ is constructed using the set-valued map $E_k \colon \mathbb{R}^n \times \mathbb{Q}_+ \times \mathbb{R}^q_+ \to 2^{\mathbb{M}_k}$, defined as follows:

Algorithm 5.2.1 (Map $E_k \colon \mathbb{R}^n \times \mathbb{Q}_+ \times \mathbb{R}^q_+ \to 2^{\mathbb{M}_k}$ for "Coordinate Search")

Parameter:	Base direction matrix $\widehat{D} = [+s^1\, e_1,\, -s^1\, e_1, \ldots,\, +s^n\, e_n,\, -s^n\, e_n]$.
	Vector $\mu \in \mathbb{N}^n$.
Input:	Iteration number $k \in \mathbb{N}$.
	Base point $x \in \mathbb{R}^n$.
	Mesh divider $\Delta_k \in \mathbb{Q}_+$.
Output:	Set of trial points $\mathcal{T}$.
Step 0:	Initialize $\mathcal{T} = \emptyset$.
	If $k = 0$, initialize , $\mu^i = 0$ for all $i \in \{1, \ldots, n\}$.
Step 1:	For $i = 1, \ldots, n$
	Set $\widetilde{x} = x + \Delta_k\, \widehat{D}\, e_{2\,i-1+\mu^i}$ and $\mathcal{T} \leftarrow \mathcal{T} \cup \{\widetilde{x}\}$.
	If $f^*(\widetilde{x}; \epsilon) < f^*(x; \epsilon)$
	Set $x = \widetilde{x}$.
	else
	If $\mu^i = 0$, set $\mu^i = 1$, else set $\mu^i = 0$.
	Set $\widetilde{x} = x + \Delta_k\, \widehat{D}\, e_{2\,i-1+\mu_i}$ and $\mathcal{T} \leftarrow \mathcal{T} \cup \{\widetilde{x}\}$.
	If $f^*(\widetilde{x}; \epsilon) < f^*(x; \epsilon)$
	Set $x = \widetilde{x}$.
	else
	If $\mu^i = 0$, set $\mu^i = 1$, else set $\mu^i = 0$.
	end if.
	end if.
	end for.
Step 2:	Return $\mathcal{T}$.

Thus, $E_k(x, \Delta_k, \epsilon) = \mathcal{T}$ for all $k \in \mathbb{N}$.

Remark 5.2.2 In Algorithm 5.2.1, $\mu \in \mathbb{N}^n$ contains for each coordinate direction an integer 0 or 1 that indicates whether a step in the positive or in the negative coordinate direction yield a decrease in cost in the last iteration. This reduces the number of exploration steps. □

d) Parameter Update

The point x' in Step 3 of the GPS Model Algorithm 5.1.14 corresponds to $x' \triangleq \arg\min_{x \in \mathcal{G}_k} f^*(x; \epsilon)$ in the Coordinate Search algorithm.

e) Keywords

For the GPS implementation of the Coordinate Search Algorithm, the command file (see page 94) must only contain continuous parameters.

To invoke the algorithm, the `Algorithm` section of the GenOpt command file must have the following form:

```
Algorithm{
   Main                        = GPSCoordinateSearch;
```

```
   MeshSizeDivider           = Integer;   // 1 <  MeshSizeDivider
   InitialMeshSizeExponent   = Integer;   // 0 <= InitialMeshSizeExponent
   MeshSizeExponentIncrement = Integer;   // 0 <  MeshSizeExponentIncrement
   NumberOfStepReduction     = Integer;   // 0 <  NumberOfStepReduction
}
```

The entries are defined as follows:

`Main` The name of the main algorithm.

`MeshSizeDivider` The value for $r \in \mathbb{N}$, $r > 1$, used to compute $\Delta_k \triangleq 1/r^{s_k}$ (see equation (5.6)). A common value is $r = 2$.

`InitialMeshSizeExponent` The value for $s_0 \in \mathbb{N}$ in (5.6). A common value is $s_0 = 0$.

`MeshSizeExponentIncrement` The value for $t_k \in \mathbb{N}$ (fixed for all $k \in \mathbb{N}$) in (5.6). A common value is $t_k = 1$.

`NumberOfStepReduction` The maximum number of step reductions before the algorithm stops. Thus, if we use the notation $m \triangleq$ `NumberOfStepReduction`, then we have for the last iterations $\Delta_k = 1/r^{s_0 + m\, t_k}$. A common value is $m = 4$.

5.2.2 Hooke-Jeeves Algorithm

We will now present the implementation of the Hooke-Jeeves algorithm [HJ61] with adaptive precision function evaluations using the Model GPS Algorithm 5.1.14. The modifications of Smith [Smi69], Bell and Pike [BP66] and De Vogelaere [DV68] are implemented in this algorithm.

To simplify the implementation, we assign $f^*(x;\epsilon) = \infty$ for all $x \notin \mathbf{X}$ where $\mathbf{X}$ is defined in (4.1).

a) Algorithm Parameters

The algorithm parameters $\widehat{D}$, r, s_0, and t_k are defined identical as in the Coordinate Search algorithm. See page 27.

b) Map for Exploratory Moves

To facilitate the algorithm explanation, we use the set-valued map $E_k \colon \mathbb{R}^n \times \mathbb{Q}_+ \times \mathbb{R}^q_+ \to 2^{\mathbb{M}_k}$, as defined in Algorithm 5.2.1. The map $E_k(\cdot,\cdot,\cdot)$ defines the "exploratory moves" in [HJ61], and will be used in Section c) to define the global search set map and, under conditions to be seen in Section d), the local search direction map as well.

c) Global Search Set Map

The global search set map $\gamma_k(\cdot,\cdot,\cdot)$ is defined as follows. Because $\gamma_0(\cdot,\cdot,\cdot)$ depends on x_{-1}, we need to introduce x_{-1}, which we define as $x_{-1} \triangleq x_0$.

Algorithm 5.2.3 (Global Search Set Map $\gamma_k \colon \underline{\mathbf{X}}_k \times \underline{\mathbf{\Delta}}_k \times \mathbb{R}^q_+ \to 2^{\mathbb{M}_k}$)

Map:	Map for "exploratory moves" $E_k \colon \mathbb{R}^n \times \mathbb{Q}_+ \times \mathbb{R}^q_+ \to 2^{\mathbb{M}_k}$.
Input:	Previous and current iterate, $x_{k-1} \in \mathbb{R}^n$ and $x_k \in \mathbb{R}^n$.
	Mesh divider $\Delta_k \in \mathbb{Q}_+$.
	Solver precision $\epsilon \in \mathbb{R}^q_+$.
Output:	Global search set $\mathcal{G}_k$.
Step 1:	Set $x = x_k + (x_k - x_{k-1})$.
Step 2:	Compute $\mathcal{G}_k = E_k(x, \Delta_k, \epsilon)$.
Step 3:	If $\left(\min_{x \in \mathcal{G}_k} f^*(x;\epsilon)\right) > f^*(x_k;\epsilon)$
	Set $\mathcal{G}_k \leftarrow \mathcal{G}_k \cup E_k(x_k, \Delta_k, \epsilon)$.
	end if.
Step 4:	Return $\mathcal{G}_k$.

Thus, $\gamma_k(\underline{x}_k, \underline{\Delta}_k, \epsilon) = \mathcal{G}_k$.

d) Local Search Direction Map

If the global search, as defined by Algorithm 5.2.3, has failed in reducing $f^*(\cdot;\epsilon)$, then Algorithm 5.2.3 has constructed a set $\mathcal{G}_k$ that contains the set $\{x_k + \Delta_k \widehat{D}\, e_i \mid i = 1, \ldots, 2n\}$. This is because in the evaluation of $E_k(x_k, \Delta_k, \epsilon)$, defined in Algorithm 5.2.1, all "If $f^*(\widetilde{x};\epsilon) < f^*(x;\epsilon)$" statements yield `false`, and, hence, one has constructed $\{x_k + \Delta_k \widehat{D}\, e_i \mid i = 1, \ldots, 2n\} = E_k(x_k, \Delta_k, \epsilon)$.

Because the columns of $\widehat{D}$ span $\mathbb{R}^n$ positively, it follows that the search on the set $\{x_k + \Delta_k \widehat{D}\, e_i \mid i = 1, \ldots, 2n\}$ is a local search. Hence, the constructed set

$$\mathcal{L}_k \triangleq \{x_k + \Delta_k \widehat{D}\, e_i \mid i = 1, \ldots, 2n\} \subset \mathcal{G}_k \tag{5.21}$$

is a local search set. Consequently, $f^*(\cdot;\epsilon)$ has already been evaluated at all points of $\mathcal{L}_k$ (during the construction of $\mathcal{G}_k$) and, hence, one does not need to evaluate $f^*(\cdot;\epsilon)$ again in a local search. In view of (5.12c) and (5.21), the local search direction map is given by $D_k = \delta_{\widehat{D},k}(\underline{x}_k, \underline{\Delta}_k) \triangleq \widehat{D}$.

e) Parameter Update

The point x' in Step 3 of the GPS Model Algorithm 5.1.14 corresponds to $x' \triangleq \arg\min_{x \in \mathcal{G}_k} f^*(x;\epsilon)$ in the Hooke-Jeeves algorithm. (Note that $\mathcal{L}_k \subset \mathcal{G}_k$ if a local search has been done as explained in the above paragraph.)

f) Keywords

For the GPS implementation of the Hooke-Jeeves algorithm, the command file (see page 94) must only contain continuous parameters.

To invoke the algorithm, the `Algorithm` section of the GenOpt command file must have the following form:

```
Algorithm{
   Main                      = GPSHookeJeeves;
   MeshSizeDivider           = Integer;   // bigger than 1
   InitialMeshSizeExponent   = Integer;   // bigger than or equal to 0
   MeshSizeExponentIncrement = Integer;   // bigger than 0
   NumberOfStepReduction     = Integer;   // bigger than 0
}
```

The entries are the same as for the Coordinate Search algorithm, and explained on page 28.

5.3 Discrete Armijo Gradient

The Discrete Armijo Gradient algorithm can be used to solve problem $\mathbf{P}_c$ defined in (4.2) where $f(\cdot)$ is continuously differentiable.

The Discrete Armijo Gradient algorithm approximates gradients by finite differences. It can be used for problems where the cost function is evaluated by computer code that defines a continuously differentiable function but for which obtaining analytical expressions for the gradients is impractical or impossible.

Since the Discrete Armijo Gradient algorithm is sensitive to discontinuities in the cost function, we recommend not to use this algorithm if the simulation program contains adaptive solvers with loose precision settings, such as EnergyPlus [CLW+01]. On such functions, the algorithm is likely to fail. In Section 4.2, we recommend algorithms that are better suited for such situations.

We will now present the Discrete Armijo Gradient algorithm and the Armijo step-size subprocedure.

Algorithm 5.3.1 (Discrete Armijo Gradient Algorithm)

Data:	Initial iterate $x_0 \in \mathbf{X}$. $\alpha, \beta \in (0,1)$, $\gamma \in (0,\infty)$, $k^*, k_0 \in \mathbb{Z}$, $l_{max}, \kappa \in \mathbb{N}$ (for reseting the step-size calculation). Termination criteria $\epsilon_m, \epsilon_x \in \mathbb{R}_+$, $i_{max} \in \mathbb{N}$.
Step 0:	Initialize $i = 0$ and $m = 0$.
Step 1:	Compute the *search direction* h_i. If $\beta^m < \epsilon_m$, stop. Else, set $\epsilon = \beta^{k_0+m}$ and compute, for $j \in \{1,\dots,n\}$, $h_i^j = -\left(f(x_i + \epsilon\, e_j) - f(x_i)\right)/\epsilon$.
Step 2 :	Check descent. Compute $\Delta(x_i; h_i) = \left(f(x_i + \epsilon\, h_i) - f(x_i)\right)/\epsilon$. If $\Delta(x_i; h_i) < 0$, go to Step 3. Else, replace m by $m+1$ and go to Step 1.
Step 3 :	Line search. Use Algorithm 5.3.2 (which requires k^*, l_{max} and κ) to compute k_i. Set $$\lambda_i = \underset{\lambda \in \{\beta^{k_i}, \beta^{k_i-1}\}}{\arg\min} f(x_i + \lambda\, h_i). \qquad (5.22)$$
Step 4 :	If $f(x_i + \lambda_i\, h_i) - f(x_i) > -\gamma\, \epsilon$, replace m by $m+1$ and go to Step 1.
Step 5 :	Set $x_{i+1} = x_i + \lambda_i\, h_i$. If $\|\lambda_i\, h_i\| < \epsilon_x$, stop. Else, replace i by $i+1$ and go to Step 1.

Algorithm 5.3.2 (Armijo Step-Size Subprocedure)

Data: Iteration number $i \in \mathbb{N}$, iterate $x_i \in \mathbb{R}^n$, search direction $h_i \in \mathbb{R}^n$, $k^*, k_{i-1} \in \mathbb{Z}$, $\alpha, \beta \in (0,\ 1)$, and $\Delta(x_i; h_i) \in \mathbb{R}$ with $\Delta(x_i; h_i) < 0$, parameter for restart $l_{max}, \kappa \in \mathbb{N}$.

Step 0: Initialize $l = 0$.
If $i = 0$, set $k' = k^*$, else set $k' = k_{i-1}$.

Step 1: Replace l by $l + 1$, and test the conditions

$$f(x_i + \beta^{k'} h_i) - f(x_i) \leq \beta^{k'} \alpha \Delta(x_i; h_i), \qquad (5.23a)$$

$$f(x_i + \beta^{k'-1} h_i) - f(x_i) > \beta^{k'-1} \alpha \Delta(x_i; h_i). \qquad (5.23b)$$

Step 2: If k' satisfies (5.23a) and (5.23b), return k'.

Step 3: If k' satisfies (5.23b) but not (5.23a),
replace k' by $k' + 1$.
else,
replace k' by $k' - 1$.
If $l < l_{max}$ or $k_{i-1} \leq k^* + \kappa$, go to Step 1. Else, go to Step 4.

Step 4: Set $\mathbf{K} \triangleq \{k \in \mathbb{Z} \mid k \geq k^*\}$, and compute
$k' \triangleq \min_{k \in \mathbf{K}} \{k \mid f(x_i + \beta^k h_i) - f(x_i) \leq \beta^k \alpha \Delta(x_i; h_i)\}$.
Return k'.

Note that in Algorithm 5.3.2, as $\beta \to 1$, the number of tries to compute the Armijo step-size is likely to go to infinity. Under appropriate assumptions one can show that $\alpha = 1/2$ yields fastest convergence [Pol97].

The step-size Algorithm 5.3.2 requires often only a small number of function evaluations. However, occasionally, once a very small step-size has occurred, Algorithm 5.3.2 can trap the Discrete Armijo Gradient algorithm into using a very small step-size for all subsequent iterations. Hence, if $k_{i-1} > k^* + \kappa$, we reset the step-size by computing Step 4.

Algorithm 5.3.1 together with the step-size Algorithm 5.3.2 have the following convergence properties [Pol97].

Theorem 5.3.3 *Let $f : \mathbb{R}^n \to \mathbb{R}$ be continuously differentiable and bounded below.*

1. *If Algorithm 5.3.1 jams at x_i, cycling indefinitely in the loop defined by Steps 1-2 or in the loop defined by Steps 1-4, then $\nabla f(x_i) = 0$.*
2. *If $\{x_i\}_{i=0}^{\infty}$ is an infinite sequence constructed by Algorithm 5.3.1 and Algorithm 5.3.2 in solving* (4.2)*, then every accumulation point $\widehat{x}$ of $\{x_i\}_{i=0}^{\infty}$ satisfies $\nabla f(\widehat{x}) = 0$.*

□

Note that ϵh_i has the same units as the cost function, and the algorithm evaluates $x_i + \lambda h_i$ for some $\lambda \in \mathbb{R}_+$. Thus, the algorithm is sensitive to the scaling of the problem variables, a rather undesirable effect. Therefore, in the

implementation of Algorithm 5.3.1 and Algorithm 5.3.2, we normalize the cost function values by replacing, for all $x \in \mathbb{R}^n$, $f(x)$ by $f(x)/f(x_0)$, where x_0 is the initial iterate. Furthermore, we set $x_0 = 0$ and evaluate the cost function for the values $\chi^j + x^j\, s^j$, $j \in \{1, \ldots, n\}$, where $x^j \in \mathbb{R}$ is the j-th component of the design parameter computed in Algorithm 5.3.1 or Algorithm 5.3.2 and $\chi^j \in \mathbb{R}$ and $s^j \in \mathbb{R}$ are the setting of the parameters `Ini` and `Step`, respectively, for the j-th design parameter in the optimization command file (see page 94).

In view of the sensitivity of the Discrete Armijo Gradient algorithm to the scaling of the problem variables and the cost function values, the implementation of penalty and barrier functions may cause numerical problems if the penalty is large compared to the unpenalized cost function value.

If box-constraints for the independent parameters are specified, then the transformations (8.2) are used.

5.3.1 Keywords

For the Discrete Armijo Gradient algorithm, the command file (see page 94) must only contain continuous parameters.

To invoke the algorithm, the `Algorithm` section of the GenOpt command file must have the following form:

```
Algorithm{
   Main  = DiscreteArmijoGradient;
   Alpha = Double;     // 0 < Alpha < 1
   Beta  = Double;     // 0 < Beta  < 1
   Gamma = Double;     // 0 < Gamma
   K0    = Integer;
   KStar = Integer;
   LMax  = Integer;    // 0 <= LMax
   Kappa = Integer;    // 0 <= LMax
   EpsilonM = Double;  // 0 < EpsilonM
   EpsilonX = Double;  // 0 < EpsilonX
}
```

The entries are defined as follows:

`Main` The name of the main algorithm.

`Alpha` The variable α used in Step 1 and in Step 4 of Algorithm 5.3.2. A typical value is $\alpha = 1/2$.

`Beta` The variable β used in approximating the gradient and doing the line search. A typical value is $\beta = 0.8$.

`Gamma` The variable γ used in Step 4 of Algorithm 5.3.1 to determine whether the accuracy of the gradient approximation will be increased.

`K0` The variable k_0 that determines the initial accuracy of the gradient approximation.

`KStar` The variable k^* used to initialize the line search.

`LMax` The variable l_{max} used in Step 3 of Algorithm 5.3.2 to determine whether the line search needs to be reinitialized.

`Kappa` The variable κ used in Step 3 of Algorithm 5.3.2 to determine whether the line search needs to be reinitialized.

`EpsilonM` The variable ϵ_m used in the determination criteria $\beta^m < \epsilon_m$ in Step 1 of Algorithm 5.3.1.

`EpsilonX` The variable ϵ_x used in the determination criteria $\|\lambda_i \, h_i\| < \epsilon_x$ in Step 5 of Algorithm 5.3.1.

5.4 Particle Swarm Optimization

Particle Swarm Optimization (PSO) algorithms are population-based probabilistic optimization algorithms first proposed by Kennedy and Eberhart [EK95, KE95] to solve problem $\mathbf{P}_c$ defined in (4.2) with possibly discontinuous cost function $f\colon \mathbb{R}^n \to \mathbb{R}$. In Section 5.4.2, we will present a PSO algorithm for discrete independent variables to solve problem $\mathbf{P}_d$ defined in (4.4), and in Section 5.4.3 we will present a PSO algorithm for continuous and discrete independent variables to solve problem $\mathbf{P}_{cd}$ defined in (4.6). To avoid ambiguous notation, we always denote the dimension of the continuous independent variable by $n_c \in \mathbb{N}$ and the dimension of the discrete independent variable by $n_d \in \mathbb{N}$.

PSO algorithms exploit a set of potential solutions to the optimization problem. Each potential solution is called a *particle*, and the set of potential solutions in each iteration step is called a *population*. PSO algorithms are global optimization algorithms and do not require nor approximate gradients of the cost function. The first population is typically initialized using a random number generator to spread the particles uniformly in a user-defined hypercube. A particle update equation that is modeled on the social behavior of members of bird flocks or fish schools determines the location of each particle in the next generation.

A survey of PSO algorithms can be found in Eberhart and Shi [ES01]. Laskari et. al. present a PSO algorithm for minimax problems [LPV02b] and for integer programming [LPV02a]. In [PV02a], Parsopoulos and Vrahatis discuss the implementation of inequality and equality constraints to solve problem $\mathbf{P}_{cg}$ defined in (4.3).

We first discuss the case where the independent variable is continuous, i.e., the case of problem $\mathbf{P}_c$ defined in (4.2).

5.4.1 PSO for Continuous Variables

We will first present the initial version of the PSO algorithm which is the easiest to understand.

In the initial version of the PSO algorithm [EK95, KE95], the update equation for the particle location is as follows: Let $k \in \mathbb{N}$ denote the generation number, let $n_P \in \mathbb{N}$ denote the number of particles in each generation, let $x_i(k) \in \mathbb{R}^{n_c}$, $i \in \{1, \ldots, n_P\}$, denote the i-th particle of the k-th generation, let $v_i(k) \in \mathbb{R}^{n_c}$ denote its velocity, let $c_1, c_2 \in \mathbb{R}_+$ and let $\rho_1(k), \rho_2(k) \sim U(0,1)$ be uniformly distributed random numbers between 0 and 1. Then, the update equation is, for all $i \in \{1, \ldots, n_P\}$ and all $k \in \mathbb{N}$,

$$\begin{aligned} v_i(k+1) &= v_i(k) + c_1\,\rho_1(k)\,\big(p_{l,i}(k) - x_i(k)\big) \\ &\quad + c_2\,\rho_2(k)\,\big(p_{g,i}(k) - x_i(k)\big), && \text{(5.24a)} \\ x_i(k+1) &= x_i(k) + v_i(k+1), && \text{(5.24b)} \end{aligned}$$

where $v_i(0) \triangleq 0$ and

$$p_{l,i}(k) \triangleq \underset{x \in \{x_i(j)\}_{j=0}^{k}}{\arg\min} f(x), \tag{5.25a}$$

$$p_{g,i}(k) \triangleq \underset{x \in \{\{x_i(j)\}_{j=0}^{k}\}_{i=1}^{n_P}}{\arg\min} f(x). \tag{5.25b}$$

Thus, $p_{l,i}(k)$ is the location that for the i-th particle yields the lowest cost over all generations, and $p_{g,i}(k)$ is the location of the best particle over all generations. The term $c_1\ \rho_1(k)\,(p_{l,i}(k) - x_i(k))$ is associated with cognition since it takes into account the particle's own experience, and the term $c_2\,\rho_2(k)\,(p_{g,i}(k) - x_i(k))$ is associated with social interaction between the particles. In view of this similarity, c_1 is called *cognitive acceleration constant* and c_2 is called *social acceleration constant*.

a) Neighborhood Topology

The minimum in (5.25b) need not be taken over all points in the population. The set of points over which the minimum is taken is defined by the *neighborhood topology*. In PSO, the neighborhood topologies are usually defined using the particle index, and not the particle location. We will use the *lbest*, *gbest*, and the *von Neumann* neighborhood topology, which we will now define.

In the *lbest* topology of size $l \in \mathbb{N}$, with $l > 1$, the neighborhood of a particle with index $i \in \{1, \ldots, n_P\}$ consist of all particles whose index are in the set

$$\mathcal{N}_i \triangleq \{i - l, \ldots i, \ldots, i + l\}, \tag{5.26a}$$

where we assume that the indices wrap around, i.e., we replace -1 by $n_P - 1$, replace -2 by $n_P - 2$, etc.

In the *gbest* topology, the neighborhood contains all points of the population, i.e.,

$$\mathcal{N}_i \triangleq \{1, \ldots, n_P\}, \tag{5.26b}$$

for all $i \in \{1, \ldots, n_P\}$.

For the *von Neumann* topology, consider a 2-dimensional lattice, with the lattice points enumerated as shown in Figure 5.4. We will use the von Neumann topology of range 1, which is defined, for $i, j \in \mathbb{Z}$, as the set of points whose indices belong to the set

$$\mathcal{N}^v_{(i,j)} \triangleq \left\{ (k,l) \;\middle|\; |k-i| + |l-j| \leq 1,\ k,l \in \mathbb{Z} \right\}. \tag{5.26c}$$

The gray points in Figure 5.4 are $\mathcal{N}^v_{(1,2)}$. For simplicity, we round in GenOpt the user-specified number of particles $n'_P \in \mathbb{N}$ to the next biggest integer n_P

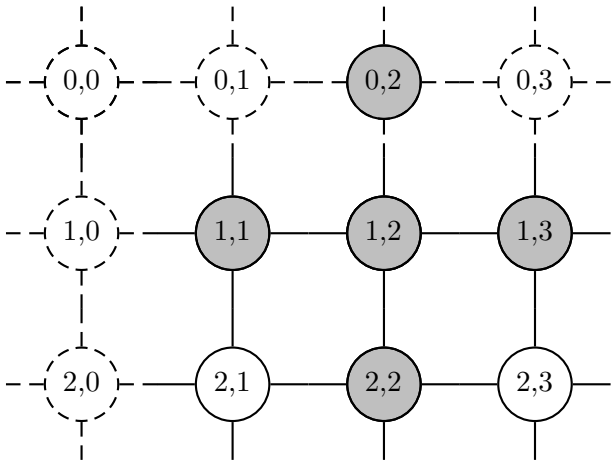


Figure 5.4: Section of a 2-dimensional lattice of particles with $\sqrt{n}_P \geq 3$. The particles belonging to the von Neumann neighborhood $\mathcal{N}^v_{(1,2)}$ with range 1, defined in (5.26c), are colored gray. Indicated by dashes are the particles that are generated by wrapping the indices.

such that $\sqrt{n}_P \in \mathbb{N}$ and $n_P \geq n'_P$.[2] Then, we can wrap the indices by replacing, for $k \in \mathbb{Z}$, $(0,k)$ by $(\sqrt{n}_P, k)$, $(\sqrt{n}_P + 1, k)$ by $(1,k)$, and similarly by replacing $(k,0)$ by $(k, \sqrt{n}_P)$ and $(k, \sqrt{n}_P + 1)$ by $(k,1)$. Then, a particle with indices (k,l), with $1 \leq k \leq \sqrt{n}_P$ and $1 \leq l \leq \sqrt{n}_P$, has in the PSO algorithm the index $i = (k-1)\sqrt{n}_P + l$, and hence $i \in \{1, \ldots, n_P\}$.

Kennedy and Mendes [KM02] show that greater connectivity of the particles speeds up convergence, but it does not tend to improve the population's ability to discover the global optimum. Best performance has been achieved with the von Neumann topology, whereas neither the *gbest* nor the *lbest* topology seemed especially good in comparison with other topologies.

Carlisle and Dozier [CD01] achieve on unimodal and multi-modal functions for the *gbest* topology better results than for the *lbest* topology.

b) Model PSO Algorithm

We will now present the Model PSO Algorithm that is implemented in GenOpt.

[2] In principle, the lattice need not be a square, but we do not see any computational disadvantage of selecting a square lattice.

Algorithm 5.4.1 (Model PSO Algorithm for Continuous Variables)

Data:	Constraint set $\mathbf{X}$, as defined in (4.1), but with finite lower and upper bound for each independent variable. Initial iterate $x_0 \in \mathbf{X}$. Number of particles $n_P \in \mathbb{N}$ and number of generations $n_G \in \mathbb{N}$.
Step 0:	Initialize $k = 0$, $x_0(0) = x_0$ and the neighborhoods $\{\mathcal{N}_i\}_{i=1}^{n_P}$.
Step 1:	Initialize $\{x_i(0)\}_{i=2}^{n_P} \subset \mathbf{X}$ randomly distributed.
Step 2:	For $i \in \{1, \ldots, n_P\}$, determine the local best particles $p_{l,i}(k) \triangleq \underset{x \in \{x_i(m)\}_{m=0}^{k}}{\arg\min} f(x)$ (5.27a) and the global best particle $p_{g,i}(k) \triangleq \underset{x \in \{x_j(m) \mid j \in \mathcal{N}_i\}_{m=0}^{k}}{\arg\min} f(x).$ (5.27b)
Step 3:	Update the particle location $\{x_i(k+1)\}_{i=1}^{n_P} \subset \mathbf{X}$.
Step 4:	If $k = n_G$, stop. Else, go to Step 2.
Step 5:	Replace k by $k+1$, and go to Step 1.

We will now discuss the different implementations of the Model PSO Algorithm 5.4.1 in GenOpt.

c) Particle Update Equation

(i) Version with Inertia Weight Eberhart and Shi [SE98, SE99] introduced an *inertia weight* $w(k)$ which improves the performance of the original PSO algorithm. In the version with inertia weight, the particle update equation is, for all $i \in \{1, \ldots, n_P\}$, for $k \in \mathbb{N}$ and $x_i(k) \in \mathbb{R}^{n_c}$, with $v_i(0) = 0$,

$$\begin{aligned} \widehat{v}_i(k+1) &= w(k)\, v_i(k) + c_1\, \rho_1(k) \left(p_{l,i}(k) - x_i(k)\right) \\ &\quad + c_2\, \rho_2(k) \left(p_{g,i}(k) - x_i(k)\right), && (5.28a) \\ v_i^j(k+1) &= \operatorname{sign}(\widehat{v}_i^j(k+1))\, \min\{|\widehat{v}_i^j(k+1)|, v_{max}^j\}, \\ &\quad j \in \{1, \ldots, n_c\}, && (5.28b) \\ x_i(k+1) &= x_i(k) + v_i(k+1), && (5.28c) \end{aligned}$$

where

$$v_{max}^j \triangleq \lambda\, (u^j - l^j), \tag{5.28d}$$

with $\lambda \in \mathbb{R}_+$, for all $j \in \{1, \ldots, n_c\}$, and $l, u \in \mathbb{R}^{n_c}$ are the lower and upper bound of the independent variable. A common value is $\lambda = 1/2$. In GenOpt, if $\lambda \leq 0$, then no velocity clamping is used, and hence, $v_i^j(k+1) = \widehat{v}_i^j(k+1)$, for all $k \in \mathbb{N}$, all $i \in \{1, \ldots, n_P\}$ and all $j \in \{1, \ldots, n_c\}$.

We compute the inertia weight as

$$w(k) = w_0 - \frac{k}{K}(w_0 - w_1), \tag{5.28e}$$

where $w_0 \in \mathbb{R}$ is the initial inertia weight, $w_1 \in \mathbb{R}$ is the inertia weight for the last generation, with $0 \leq w_1 \leq w_0$, and $K \in \mathbb{N}$ is the maximum number of generations. $w_0 = 1.2$ and $w_1 = 0$ can be considered as good choices [PV02b].

(ii) Version with Constriction Coefficient Clerc and Kennedy [CK02] introduced a version with a constriction coefficient that reduces the velocity. In their Type 1" implementation, the particle update equation is, for all $i \in \{1, \ldots, n_P\}$, for $k \in \mathbb{N}$ and $x_i(k) \in \mathbb{R}^{n_c}$, with $v_i(0) = 0$,

$$\begin{aligned}
\widehat{v}_i(k+1) &= \chi(\kappa,\varphi)\left(v_i(k) + c_1\,\rho_1(k)\left(p_{l,i}(k) - x_i(k)\right)\right. \\
&\quad \left. + c_2\,\rho_2(k)\left(p_{g,i}(k) - x_i(k)\right)\right), && (5.29a)\\
v_i^j(k+1) &= \operatorname{sign}(\widehat{v}_i^j(k+1))\,\min\{|\widehat{v}_i^j(k+1)|, v_{max}^j\}, \\
&\quad j \in \{1, \ldots, n_c\}, && (5.29b)\\
x_i(k+1) &= x_i(k) + v_i(k+1), && (5.29c)
\end{aligned}$$

where

$$v_{max}^j \triangleq \lambda\,(u^j - l^j), \tag{5.29d}$$

is as in (5.28d).

In (5.29a), $\chi(\kappa, \varphi)$ is called *constriction coefficient*, defined as

$$\chi(\kappa,\varphi) \triangleq \begin{cases} \frac{2\,\kappa}{|2-\varphi-\sqrt{\varphi^2-4\,\varphi}|}, & \text{if } \varphi > 4, \\ \kappa, & \text{otherwise,} \end{cases} \tag{5.29e}$$

where $\varphi \triangleq c_1 + c_2$ and $\kappa \in (0, 1]$ control how fast the population collapses into a point. If $\kappa = 1$, the space is thoroughly searched, which yields slower convergence.

Equation (5.29) can be used with or without velocity clamping (5.29b). If velocity clamping (5.29b) is used, Clerc and Kennedy use $\varphi = 4.1$, otherwise they use $\varphi = 4$. In either case, they set $c_1 = c_2 = \varphi/2$ and a population size of $n_P = 20$.

Carlisle and Dozier [CD01] recommend the settings $n_P = 30$, no velocity clamping, $\kappa = 1$, $c_1 = 2.8$ and $c_2 = 1.3$.

Kennedy and Eberhart [KES01] report that using velocity clamping (5.29b) and a constriction coefficient shows faster convergence for some test problems compared to using an inertia weight, but the algorithm tends to get stuck in local minima.

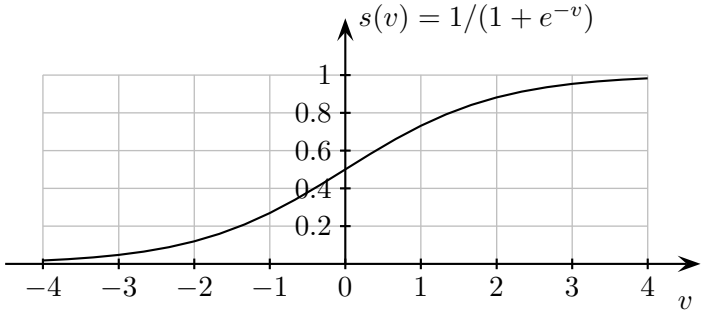


Figure 5.5: Sigmoid function.

5.4.2 PSO for Discrete Variables

Kennedy and Eberhart [KE97] introduced a binary version of the PSO algorithm to solve problem $\mathbf{P}_d$ defined in (4.4).

The binary PSO algorithm encodes the discrete independent variables in a string of binary numbers and then operates with this binary string. For some $i \in \{1, \ldots, n_d\}$, let $x_i \in \mathbb{N}$ be the component of a discrete independent variable, and let $\psi_i \in \{0,1\}^{m_i}$ be its binary representation (with $m_i \in \mathbb{N}_+$ bits), obtained using Gray encoding [PFTV93], and let $\pi_{l,i}(k)$ and $\pi_{g,i}(k)$ be the binary representation of $p_{l,i}(k)$ and $p_{g,i}(k)$, respectively, where $p_{l,i}(k)$ and $p_{g,i}(k)$ are defined in (5.27).

Then, for $i \in \{1, \ldots, n_d\}$ and $j \in \{1, \ldots, m_i\}$ we initialize randomly $\psi_i^j(0) \in \{0,1\}$, and compute, for $k \in \mathbb{N}$,

$$\begin{aligned} \widehat{v}_i^j(k+1) &= v_i^j(k) + c_1\,\rho_1(k)\left(\pi_{l,i}^j(k) - \psi_i^j(k)\right) \\ &\quad + c_2\,\rho_2(k)\left(\pi_{g,i}^j(k) - \psi_i^j(k)\right)\big), \qquad (5.30a) \\ v_i^j(k+1) &= \operatorname{sign}(\widehat{v}_i^j(k+1))\,\min\{|\widehat{v}_i^j(k+1)|, v_{max}\}, \qquad (5.30b) \\ \psi_i^j(k+1) &= \begin{cases} 0, & \text{if } \rho_{i,j}(k) \geq s\big(v_i^j(k+1)\big), \\ 1, & \text{otherwise}, \end{cases} \qquad (5.30c) \end{aligned}$$

where

$$s(v) \triangleq \frac{1}{1+e^{-v}} \qquad (5.30d)$$

is the sigmoid function shown in Fig. 5.5 and $\rho_{i,j}(k) \sim U(0,1)$, for all $i \in \{1, \ldots, n_d\}$ and for all $j \in \{1, \ldots, m_i\}$.

In (5.30b), $v_{max} \in \mathbb{R}_+$ is often set to 4 to prevent a saturation of the sigmoid function, and $c_1, c_2 \in \mathbb{R}_+$ are often such that $c_1 + c_2 = 4$ (see [KES01]).

Notice that $s(v) \to 0.5$, as $v \to 0$, and consequently the probability of flipping a bit goes to 0.5. Thus, in the binary PSO, a small v_{max} causes a large exploration, whereas in the continuous PSO, a small v_{max} causes a small exploration of the search space.

Any of the above neighborhood topologies can be used, and Model Algorithm 5.4.1 applies if we replace the constraint set $\mathbf{X}$ by the user-specified set $\mathbf{X}_d \subset \mathbb{Z}^{n_d}$.

5.4.3 PSO for Continuous and Discrete Variables

For problem $\mathbf{P}_{cd}$ defined in (4.6), we treat the continuous independent variables as in (5.28) or (5.29), and the discrete independent variables as in (5.30). Any of the above neighborhood topologies can be used, and Model Algorithm 5.4.1 applies if we define the constraint set $\mathbf{X}$ as in (4.5).

5.4.4 PSO on a Mesh

We now present a modification to the previously discussed PSO algorithms. For evaluating the cost function, we will modify the continuous independent variables such that they belong to a fixed mesh in $\mathbb{R}^{n_c}$. Since PSO algorithms typically cluster points during the last iterations, this reduces in many cases the number of simulation calls during the optimization. The modification is done by replacing the cost function $f\colon \mathbb{R}^{n_c} \times \mathbb{Z}^{n_d} \to \mathbb{R}$ in Model Algorithm 5.4.1 as follows: Let $x_0 \triangleq (x_{c,0}, x_{d,0}) \in \mathbb{R}^{n_c} \times \mathbb{Z}^{n_c}$ denote the initial iterate, let $\mathbf{X}_c$ be the feasible set for the continuous independent variables defined in (4.5b), let $r, s \in \mathbb{N}$, with $r > 1$, be user-specified parameters, let

$$\Delta \triangleq \frac{1}{r^s} \tag{5.31}$$

and let the mesh be defined as

$$\mathbb{M}(x_{c,0}, \Delta, s) \triangleq \{x_{c,0} + m\, \Delta\, s^i\, e_i \mid i \in \{1, \ldots, n_c\},\ m \in \mathbb{Z}\}, \tag{5.32}$$

where $s \in \mathbb{R}^{n_c}$ is equal to the value defined by the variable `Step` in GenOpt's command file (see page 94). Then, we replace $f(\cdot, \cdot)$ by $\widehat{f}\colon \mathbb{R}^{n_c} \times \mathbb{Z}^{n_d} \times \mathbb{R}^{n_c} \times \mathbb{R} \times \mathbb{R}^{n_c} \to \mathbb{R}$, defined by

$$\widehat{f}(\widehat{x}_c, x_d; x_{c,0}, \Delta, s) \triangleq f(\widehat{x}_c, x_d), \tag{5.33}$$

where

$$\widehat{x}_c \triangleq \arg\min\{\|x_c - x\| \mid x \in \mathbb{M}(x_0, \Delta, s) \cap \mathbf{X}_c\}. \tag{5.34}$$

Thus, the continuous independent variables are replaced by the closest feasible mesh point, and the discrete independent variables remain unchanged.

Good numerical results have been obtained by selecting $s \in \mathbb{R}^{n_c}$ and $r, s \in \mathbb{N}$ such that about 50 to 100 mesh points are located along each coordinate direction.

5.4.5 Population Size and Number of Generations

Parsopoulos and Vrahatis [PV02b] use for $x \in \mathbb{R}^{n_c}$ a population size of about $5\,n$ up to $n = 15$. For $n \approx 10 \ldots 20$, they use $n_P \approx 10\,n$. They set the number of generations to $n_G = 1000$ up to $n = 20$ and to $n_G = 2000$ for $n = 30$.

Van den Bergh and Engelbrecht [vdBE01] recommend using more than 20 particles and 2000 to 5000 generations.

Kennedy and Eberhart [KES01] use, for test cases with the *lbest* neighborhood topology of size $l = 2$ and $n = 2$ and $n = 30$, a population size of $n_P = 20 \ldots 30$. They report that $10 \ldots 50$ particles usually work well. As a rule of thumb, they recommend for the *lbest* neighborhood to select the neighborhood size such that each neighborhood consists of $10 \ldots 20\%$ of the population.

5.4.6 Keywords

For the Particle Swarm algorithm, the command file (see page 94) can contain continuous and discrete independent variables.

The different specifications for the `Algorithm` section of the GenOpt command file are as follows:

PSO algorithm with inertia weight:

```
Algorithm{
  Main                      = PSOIW;
  NeighborhoodTopology      = gbest | lbest | vonNeumann;
  NeighborhoodSize          = Integer;  // 0 < NeighborhoodSize
  NumberOfParticle          = Integer;
  NumberOfGeneration        = Integer;
  Seed                      = Integer;
  CognitiveAcceleration     = Double;   // 0 < CognitiveAcceleration
  SocialAcceleration        = Double;   // 0 < SocialAcceleration
  MaxVelocityGainContinuous = Double;
  MaxVelocityDiscrete       = Double;   // 0 < MaxVelocityDiscrete
  InitialInertia            = Double;   // 0 < InitialInertia
  FinalInertia              = Double;   // 0 < FinalInertia
}
```

PSO algorithm with constriction coefficient:

```
Algorithm{
  Main                      = PSOCC;
  NeighborhoodTopology      = gbest | lbest | vonNeumann;
  NeighborhoodSize          = Integer;  // 0 < NeighborhoodSize
  NumberOfParticle          = Integer;
  NumberOfGeneration        = Integer;
  Seed                      = Integer;
```

```
  CognitiveAcceleration     = Double;   // 0 < CognitiveAcceleration
  SocialAcceleration        = Double;   // 0 < SocialAcceleration
  MaxVelocityGainContinuous = Double;
  MaxVelocityDiscrete       = Double;   // 0 < MaxVelocityDiscrete
  ConstrictionGain          = Double;   // 0 < ConstrictionGain <= 1
}
```

PSO algorithm with constriction coefficient and continuous independent variables restricted to a mesh:

```
Algorithm{
  Main                      = PSOCCMesh;
  NeighborhoodTopology      = gbest | lbest | vonNeumann;
  NeighborhoodSize          = Integer;  // 0 < NeighborhoodSize
  NumberOfParticle          = Integer;
  NumberOfGeneration        = Integer;
  Seed                      = Integer;
  CognitiveAcceleration     = Double;   // 0 < CognitiveAcceleration
  SocialAcceleration        = Double;   // 0 < SocialAcceleration
  MaxVelocityGainContinuous = Double;
  MaxVelocityDiscrete       = Double;   // 0 < MaxVelocityDiscrete
  ConstrictionGain          = Double;   // 0 < ConstrictionGain <= 1
  MeshSizeDivider           = Integer;  // 1 < MeshSizeDivider
  InitialMeshSizeExponent   = Integer;  // 0 <= InitialMeshSizeExponent
}
```

The entries that are common to all implementations are defined as follows:

`Main` The name of the main algorithm. The implementation `PSOIW` uses the location update equation (5.28) for the continuous independent variables, and the implementation `PSOCC` uses (5.29) for the continuous independent variables. All implementations use (5.30) for the discrete independent variables.

`NeighborhoodTopology` This entry defines what neighborhood topology is being used.

`NeighborhoodSize` This entry is equal to l in (5.26). For the *gbest* neighborhood topology, the value of `NeighborhoodSize` will be ignored.

`NumberOfParticle` This is equal to the variable $n_P \in \mathbb{N}$.

`NumberOfGeneration` This is equal to the variable $n_G \in \mathbb{N}$ in Algorithm 5.4.1.

`Seed` This value is used to initialize the random number generator.

`CognitiveAcceleration` This is equal to the variable $c_1 \in \mathbb{R}_+$.

`SocialAcceleration` This is equal to the variable $c_2 \in \mathbb{R}_+$.

`MaxVelocityGainContinuous` This is equal to the variable $\lambda \in \mathbb{R}_+$ in (5.28d) and in (5.29d). If `MaxVelocityGainContinuous` is set to zero or to a negative value, then no velocity clamping is used, and hence, $v_i^j(k+1) = \widehat{v}_i^j(k+1)$, for all $k \in \mathbb{N}$, all $i \in \{1, \ldots, n_P\}$ and all $j \in \{1, \ldots, n_c\}$.

`MaxVelocityDiscrete` This is equal to the variable $v_{max} \in \mathbb{R}_+$ in (5.30b).

For the `PSOIW` implementation, following additional entries must be specified:

`InitialInertia` This is equal to $w_0 \in \mathbb{R}_+$ in (5.28e).

`FinalInertia` This is equal to $w_1 \in \mathbb{R}_+$ in (5.28e).

For the `PSOCC` implementation, following additional entries must be specified:

`ConstrictionGain` This is equal to $\kappa \in (0, 1]$ in (5.29e).

Notice that for discrete independent variables, the entries of `InitialInertia`, `FinalInertia`, and `ConstrictionGain` are ignored.

For the `PSOCCMesh` implementation, following additional entries must be specified:

`MeshSizeDivider` This is equal to $r \in \mathbb{N}$, with $r > 1$, used in (5.31).

`InitialMeshSizeExponent` This is equal to $s \in \mathbb{N}$ used in (5.31).

5.5 Hybrid Generalized Pattern Search Algorithm with Particle Swarm Optimization Algorithm

This hybrid global optimization algorithm can be used to solve problem $\mathbf{P}_c$ defined in (4.2) and problem $\mathbf{P}_{cd}$ defined in (4.6). Problem $\mathbf{P}_{cg}$ defined in (4.3) and problem $\mathbf{P}_{cdg}$ defined in (4.7) can be solved if the constraint functions $g(\cdot)$ are implemented as described in Section 8.2.

This hybrid global optimization algorithm starts by doing a Particle Swarm Optimization (PSO) on a mesh, as described in Section 5.4.4, for a user-specified number of generations $n_G \in \mathbb{N}$. Afterwards, it initializes the Hooke-Jeeves Generalized Pattern Search (GPS) algorithm, described in Section 5.2.2, using the continuous independent variables of the particle with the lowest cost function value. If the optimization problem has continuous and discrete independent variables, then the discrete independent variables will for the GPS algorithm be fixed at the value of the particle with the lowest cost function value.

We will now explain the hybrid algorithm for the case where all independent variables are continuous, and then for the case with mixed continuous and discrete independent variables. Throughout this section, we will denote the dimension of the continuous independent variables by $n_c \in \mathbb{N}$ and the dimension of the discrete independent variables by $n_d \in \mathbb{N}$.

5.5.1 Hybrid Algorithm for Continuous Variables

We will now discuss the hybrid algorithm to solve problem $\mathbf{P}_c$ defined in (4.2). However, we require the constraint set $\mathbf{X} \subset \mathbb{R}^{n_c}$ defined in (4.1) to have finite lower and upper bounds $l^i, u^i \in \mathbb{R}$, for all $i \in \{1, \ldots, n_c\}$.

First, we run the PSO algorithm 5.4.1, with user-specified initial iterate $x_0 \in \mathbf{X}$ for a user-specified number of generations $n_G \in \mathbb{N}$ on the mesh defined in (5.32). Afterwards, we run the GPS algorithm 5.1.14 where the initial iterate x_0 is equal to the location of the particle with the lowest cost function value, i.e.,

$$x_0 \triangleq p \triangleq \underset{x \in \{x_j(k) \mid j \in \{1,\ldots,n_P\},\ k \in \{1,\ldots,n_G\}\}}{\arg\min} f(x), \tag{5.35}$$

where $n_P \in \mathbb{N}$ denotes the number of particles and $x_j(k)$, $j \in \{1, \ldots, n_P\}$, $k \in \{1, \ldots, n_G\}$ are as in Algorithm 5.4.1.

Since the PSO algorithm terminates after a finite number of iterations, all convergence results of the GPS algorithm hold. In particular, if the cost function is once continuously differentiable, then the hybrid algorithm constructs accumulation points that are feasible stationary points of problem (4.2) (see Theorem 5.1.24).

Since the PSO algorithm is a global optimization algorithm, the hybrid algorithm is, compared to the Hooke-Jeeves algorithm, less likely to be attracted by a local minimum that is not global. Thus, the hybrid algorithm combines the global features of the PSO algorithm with the provable convergence properties of the GPS algorithm.

If the cost function is discontinuous, then the hybrid algorithm is, compared to the Hooke-Jeeves algorithm, less likely to jam at a discontinuity far from a solution.

5.5.2 Hybrid Algorithm for Continuous and Discrete Variables

For problem $\mathbf{P}_{cd}$ defined in (4.6) with continuous and discrete independent variables, we run the PSO algorithm 5.4.1, with user-specified initial iterate $x_0 \in \mathbf{X} \triangleq \mathbf{X}_c \times \mathbf{X}_d \subset \mathbb{R}^{n_c} \times \mathbb{Z}^{n_d}$ for a user-specified number of generations $n_G \in \mathbb{N}$, where the continuous independent variables are restricted to the mesh defined in (5.32). We require the constraint set $\mathbf{X}_c \subset \mathbb{R}^{n_c}$ defined in (4.5b) to have finite lower and upper bounds $l^i, u^i \in \mathbb{R}$, for all $i \in \{1, \ldots, n_c\}$.

Afterwards, we run the GPS algorithm 5.1.14, where the initial iterate $x_0 \in \mathbf{X}_c$ is equal to $p_c \in \mathbf{X}_c$, which we define as the continuous independent variables of the particle with the lowest cost function value, i.e., $p \triangleq (p_c, p_d) \in \mathbf{X}_c \times \mathbf{X}_d$, where p is defined in (5.35). In the GPS algorithm, we fix the discrete components at $p_d \in \mathbf{X}_d$ for all iterations. Thus, we use the GPS algorithm to refine the continuous components of the independent variables, and fix the discrete components of the independent variables.

5.5.3 Keywords

For this algorithm, the command file (see page 94) can contain continuous and discrete independent variables. It must contain at least one continuous parameter.

The specifications of the `Algorithm` section of the GenOpt command file is as follows:

Note that the first entries are as for the PSO algorithm on page 44 and the last entries are as for GPS implementation of the Hooke-Jeeves algorithm on page 30.

```
Algorithm{
  Main                    = GPSPSOCCHJ;
  NeighborhoodTopology    = gbest | lbest | vonNeumann;
  NeighborhoodSize        = Integer;  // 0 <  NeighborhoodSize
  NumberOfParticle        = Integer;
  NumberOfGeneration      = Integer;
  Seed                    = Integer;
```

```
  CognitiveAcceleration     = Double;   // 0 <  CognitiveAcceleration
  SocialAcceleration        = Double;   // 0 <  SocialAcceleration
  MaxVelocityGainContinuous = Double;
  MaxVelocityDiscrete       = Double;   // 0 <  MaxVelocityDiscrete
  ConstrictionGain          = Double;   // 0 <  ConstrictionGain <= 1
  MeshSizeDivider           = Integer;  // 1 <  MeshSizeDivider
  InitialMeshSizeExponent   = Integer;  // 0 <= InitialMeshSizeExponent
  MeshSizeExponentIncrement = Integer;  // 0 <  MeshSizeExponentIncrement
  NumberOfStepReduction     = Integer;  // 0 <  NumberOfStepReduction
}
```

The entries are defined as follows:

`Main` The name of the main algorithm.

`NeighborhoodTopology` This entry defines what neighborhood topology is being used.

`NeighborhoodSize` This entry is equal to l in (5.26). For the *gbest* neighborhood topology, the value of `NeighborhoodSize` will be ignored.

`NumberOfParticle` This is equal to the variable $n_P \in \mathbb{N}$.

`NumberOfGeneration` This is equal to the variable $n_G \in \mathbb{N}$ in Algorithm 5.4.1.

`Seed` This value is used to initialize the random number generator.

`CognitiveAcceleration` This is equal to the variable $c_1 \in \mathbb{R}_+$ used by the PSO algorithm.

`SocialAcceleration` This is equal to the variable $c_2 \in \mathbb{R}_+$ used by the PSO algorithm.

`MaxVelocityGainContinuous` This is equal to the variable $\lambda \in \mathbb{R}_+$ in (5.28d) and in (5.29d). If `MaxVelocityGainContinuous` is set to zero or to a negative value, then no velocity clamping is used, and hence, $v_i^j(k+1) = \widehat{v}_i^j(k+1)$, for all $k \in \mathbb{N}$, all $i \in \{1, \ldots, n_P\}$ and all $j \in \{1, \ldots, n_c\}$.

`MaxVelocityDiscrete` This is equal to the variable $v_{max} \in \mathbb{R}_+$ in (5.30b).

`ConstrictionGain` This is equal to $\kappa \in (0, 1]$ in (5.29e).

`MeshSizeDivider` This is equal to $r \in \mathbb{N}$, with $r > 1$, used by the PSO algorithm in (5.31) and used by the GPS algorithm to compute $\Delta_k \triangleq 1/r^{s_k}$ (see equation (5.6)). A common value is $r = 2$.

`InitialMeshSizeExponent` This is equal to $s \in \mathbb{N}$ used by the PSO algorithm in (5.31) and used by the GPS algorithm in (5.6). A common value is $s_0 = 0$.

`MeshSizeExponentIncrement` The value for $t_k \in \mathbb{N}$ (fixed for all $k \in \mathbb{N}$) used by the GPS algorithm in (5.6). A common value is $t_k = 1$.

`NumberOfStepReduction` The maximum number of step reductions before the GPS algorithm stops. Thus, if we use the notation $m \triangleq$ `NumberOfStepReduction`, then we have for the last iterations $\Delta_k = 1/r^{s_0 + m\, t_k}$. A common value is $m = 4$.

5.6 Hooke-Jeeves

This algorithm is implemented for compatibility with previous GenOpt versions and is no longer supported. We recommend using the implementation of the Hooke-Jeeves algorithm described in Section 5.2.2 on page 29.

The Hooke-Jeeves pattern search algorithm [HJ61] is a derivative free optimization algorithm that can be used to solve problem $\mathbf{P}_c$ defined in (4.2) for $n > 1$. Problem $\mathbf{P}_{cg}$ defined in (4.3) can be solved by implementing constraints on the dependent parameters as described in Section 8.

For problem (4.2), if the cost function is continuously differentiable and has bounded level sets, then the Hooke-Jeeves algorithm converges to a point $x^* \in \mathbb{R}^n$ that satisfies $\|\nabla f(x^*)\| = 0$ (see [Tor97, AD03]).

Hooke and Jeeves found empirically that the number of function evaluations increases only linearly with the number of independent variables [HJ61].

5.6.1 Modifications to the Original Algorithm

Now, we explain modifications to the original algorithm of [HJ61] which are implemented in GenOpt.

Smith [Smi69] reports that applying the same step size for each variable causes some parameters to be essentially ignored during much of the search process. Therefore, Smith proposes to initialize the step size for each variable by

$$\Delta x^i = \delta |x_0^i|, \tag{5.36}$$

where $\delta > 0$ is a fraction of the initial step length and $x_0 \in \mathbb{R}^n$ is the initial iterate. In GenOpt's implementation, Δx^i is set equal to the value of the parameter `Step`, which is specified in the command file (see page 94). This allows taking the scaling of the components of the independent parameter into account.

In [HJ61], the search of the exploration move is always done first in the positive, then in the negative direction along the coordinate vectors, $e_i \in \mathbb{R}^n$, $i \in \{1, \ldots, n\}$. Bell and Pike [BP66] proposed searching first in the direction that led in the last exploration move to a reduction of the cost function. This increases the probability to reduce the cost function already by the first exploration move, thus allows skipping the second trial.

De Vogelaere [DV68] proposed changing the algorithm such that the maximum number of function evaluations cannot be exceeded, which can be the case in the original implementation.

All three modifications are implemented.

To implement the box constraints of problem $\mathbf{P}_c$ and $\mathbf{P}_{cg}$, defined in (4.2) and (4.3), respectively, we assign $f^*(x;\epsilon) = \infty$ for all $x \notin \mathbf{X}$ where $\mathbf{X}$ is defined in (4.1).

5.6.2 Algorithm Description

Hooke and Jeeves divide the algorithm in an *initial exploration* (I), a *basic iteration* (II), and a *step size reduction* (III). (I) and (II) make use of so-called *exploratory moves* to get local information about the direction in which the cost function decreases.

The exploratory moves are executed as follows (see Fig. 5.6):
Let $\Delta x^i \in \mathbb{R}$ be the step size of the i-th independent parameter, and $e_i \in \mathbb{R}^n$ the unit vector along the i-th coordinate direction. Assume we are given a base point, called the resulting base point x_r and its function value, say $f_p \triangleq f(x_r)$. Then we make a sequence of orthogonal exploratory moves. To do so, we set $i = 0$ and assign

$$x_r \leftarrow x_r + \Delta x^i \, e_i. \tag{5.37}$$

Provided that x_r is feasible, that is $x_r \in \mathbf{X}$, we evaluate the cost function and assign $f_r \leftarrow f(x_r)$. If $f_r < f_p$, then the new point becomes the resulting base point, and we assign

$$f_p \leftarrow f_r. \tag{5.38}$$

Otherwise, we assign

$$\Delta x^i \leftarrow -\Delta x^i, \tag{5.39}$$

$$x_r \leftarrow x_r + 2\,\Delta x^i \, e_i, \tag{5.40}$$

evaluate $f(x_r)$ and assign $f_r \leftarrow f(x_r)$. If this exploration reduced the cost function, we apply (5.38). Otherwise, we reset the resulting base point by assigning

$$x_r \leftarrow x_r - \Delta x^i \, e_i \tag{5.41}$$

so that the resulting base point has not been altered by the exploration in the direction along e_i. Therefore, if any of the exploration moves have been successful, we have a new resulting base point x_r and a new function value $f_p = f(x_r)$. Using the (probably new) resulting base point x_r the same procedure is repeated along the next coordinate direction (i.e., along e_{i+1},) until an exploration along all coordinate vectors e_i, $i \in \{1, \ldots, n\}$ has been done. Note that according to (5.40) Δx^i has in the next exploration move along e_i the sign that led in the last exploration to a reduction of the cost function (if any reduction was achieved).

At the end of the n exploratory moves, we have a new resulting base point x_r if and only if at least one of the exploratory moves led to a reduction of the

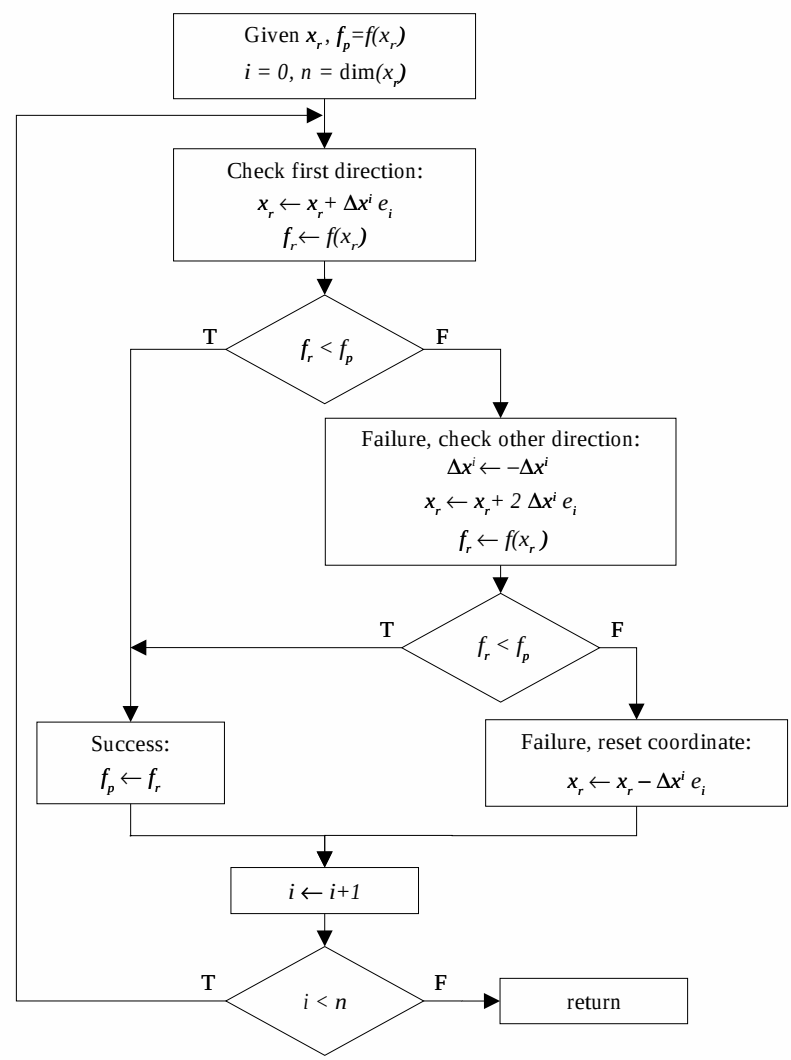


Figure 5.6: *Flow chart of the exploration move, $E(x_r, f_p)$.*

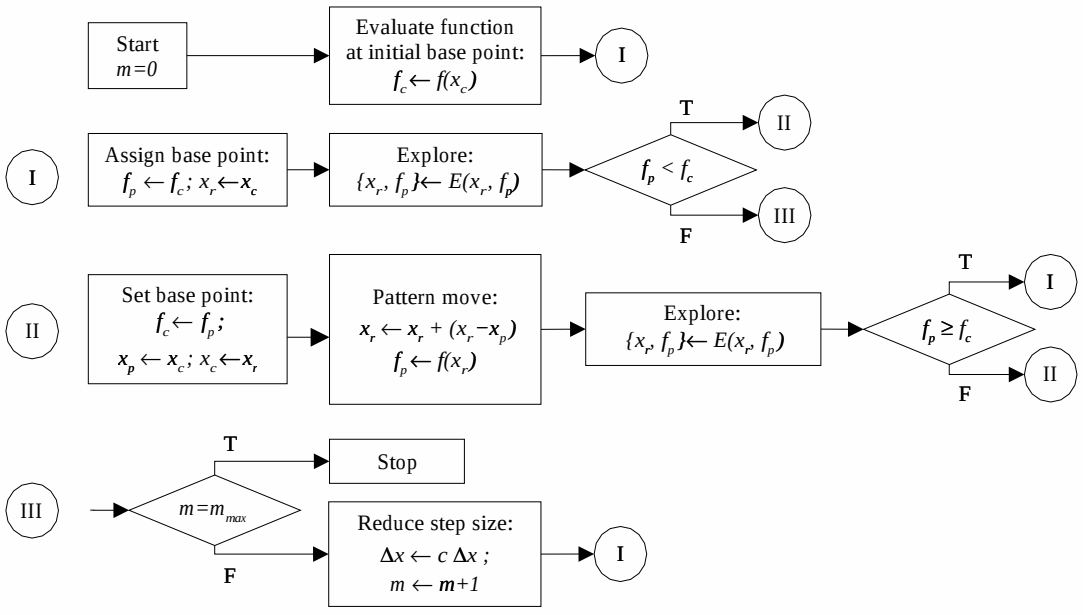


Figure 5.7: Flow chart of Hooke-Jeeves algorithm.

cost function.

(I) Initial Iteration
In the initial iteration, we have a current base point, x_c. We assign $x_r \leftarrow x_c$ and make the exploration moves around x_r. If at least one of the exploration move leads to a reduction of the cost function, then we go to the basic iteration (II), otherwise we reduce the step size according to (III).

(II) Basic Iteration
We update the function value of the base point by assigning

$$f_c \leftarrow f_p, \tag{5.42}$$

and assign to the previous base point x_p the value of the current base point x_c and to the current base point x_c the value of the resulting base point x_r, i.e.,

$$x_p \leftarrow x_c, \tag{5.43}$$
$$x_c \leftarrow x_r. \tag{5.44}$$

Then, we make a pattern move, given by

$$x_r \leftarrow x_r + (x_r - x_p). \tag{5.45}$$

Now, we assign $f_p \leftarrow f(x_r)$. Regardless of whether the pattern move leads to a reduction of the cost function, we do exploratory moves around x_r. If any

of the exploratory moves is successful, then x_r and consequently $f_p = f(x_r)$ are altered. Now, we check whether $f_p \geq f_r$. If so, the pattern move might no longer be appropriate and we do an initial step (I). Otherwise, the pattern move and the exploration steps lead to an improvement and we do a basic iteration again (II).

(III) Step Size Reduction
The relative step size for the exploration moves is reduced according to

$$\Delta x \leftarrow c\,\Delta x, \tag{5.46}$$

where $0 < c < 1$ is the constant step reduction factor. A common value for c is 0.5. x_c is considered as the minimum point and the algorithm stops if the step size has been reduced m_{max} times. m_{max} is a user input.

Further discussion of this algorithm can be found in [HJ61, Wil64, Avr76, Wal75].

5.6.3 Keywords

For the Hooke-Jeeves algorithm, the command file (see page 94) must only contain continuous parameters.

To invoke the Hooke-Jeeves algorithm, the `Algorithm` section of the GenOpt command file must have the following form:

```
Algorithm{
   Main                    = HookeJeeves;
   StepReduction           = Double;    // 0 < StepReduction
   NumberOfStepReduction   = Integer;   // 0 < NumberOfStepReduction
}
```

The entries are defined as follows:

`Main` The name of the main algorithm.

`StepReduction` The step reduction factor, c in (5.46), where $0 < c < 1$. A common value is $c = 0.5$.

`NumberOfStepReduction` This integer specifies how many times a step reduction has to be done before a point is considered as being a minimum point. `NumberOfStepReduction` is equal to the parameter m_{max} in Fig. 5.7. A common value is $m_{max} = 2$, but m_{max} depends on the step size and the required accuracy.

5.7 Simplex Algorithm of Nelder and Mead with the Extension of O'Neill

The Simplex algorithm of Nelder and Mead is a derivative free optimization algorithm. It can be used to seek a solution of problem $\mathbf{P}_c$ defined in (4.2) and problem $\mathbf{P}_{cg}$ defined in (4.3), with constraints on the dependent parameters implemented as described in Section 8. The number of independent parameters n must be larger than 1.

The Simplex algorithm constructs an n-dimensional simplex in the space that is spanned by the independent parameters. At each of the $(n+1)$ vertices of the simplex, the value of the cost function is evaluated. In each iteration step, the point with the highest value of the cost function is replaced by another point. The algorithm consists of three main operations: (a) *point reflection*, (b) *contraction of the simplex* and (c) *expansion of the simplex*.

It is known that the Simplex algorithm can fail to converge to a stationary point, even if the cost function is smooth. For an excellent discussion, see [Wri96] or [McK98, LRWW98]. The Simplex algorithm fails when the simplex collapses into a subspace of $\mathbb{R}^n$, or becomes elongated and distorted in shape. Despite its bad convergence properties, the Simplex algorithm often successfully locates a greatly improved solution with many fewer function evaluations than its competitors [Wri96].

In [McK98], McKinnon presents a strictly convex function with three continuous derivatives and a set of initial iterates for which the Simplex algorithm converges to a non-stationary point. In this example, the vertices tend to a straight line which is orthogonal to the steepest descent direction.

We will now explain the different steps of the Simplex algorithm.

5.7.1 Main Operations

The notation defined below is used in describing the main operations. The operations are illustrated in Fig. 5.8 where for simplicity a two-dimensional simplex is illustrated.

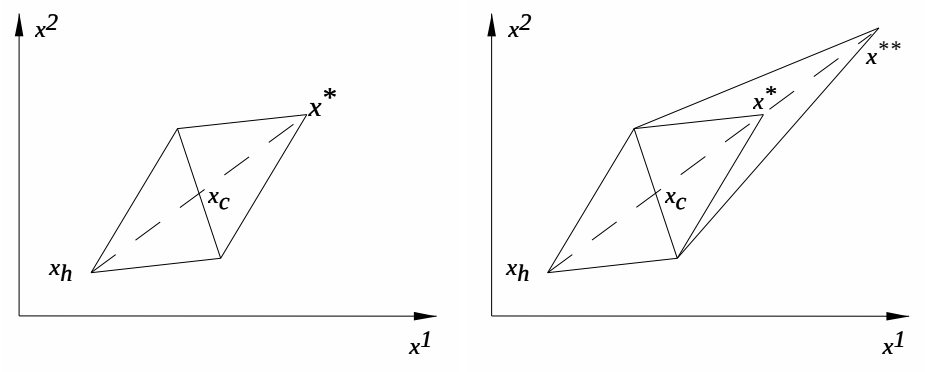
We now introduce some notation and definitions.

1. We will denote by $\mathbf{I} \triangleq \{1, \ldots, n+1\}$ the set of all vertex indices.
2. We will denote by $l \in \mathbf{I}$ the smallest index in $\mathbf{I}$ such that

$$l = \arg\min_{i \in \mathbf{I}} f(x_i). \tag{5.47a}$$

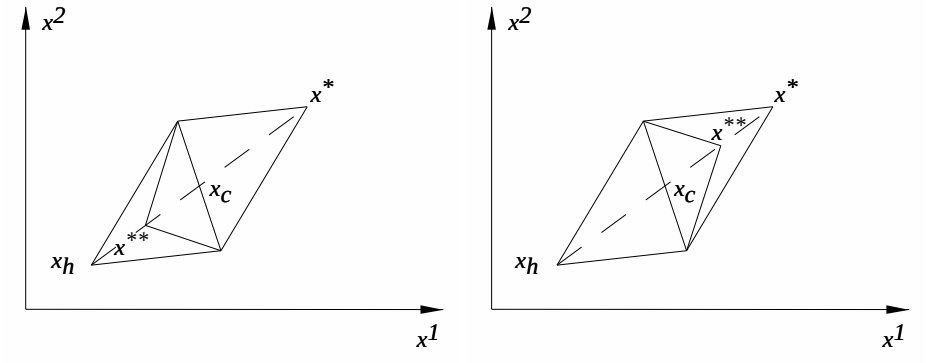
   Hence, $f(x_l) \leq f(x_i)$, for all $i \in \mathbf{I}$.
3. We will denote by $h \in \mathbf{I}$ the smallest index in $\mathbf{I}$ such that

$$h = \arg\max_{i \in \mathbf{I}} f(x_i). \tag{5.47b}$$



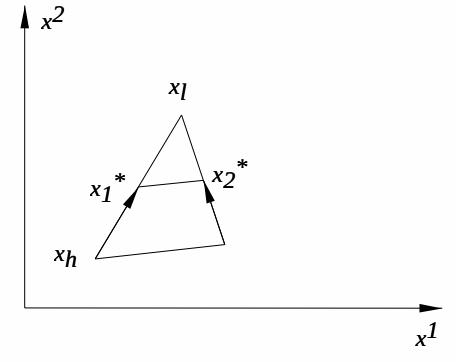
(a) Reflection.

(b) Expansion.



(c) Partial inside contraction.

(d) Partial outside contraction.



(e) Total contraction.

Figure 5.8: *Simplex operations.*

Hence, $f(x_h) \geq f(x_i)$, for all $i \in \mathbf{I}$.

4. Let x_i, for $i \in \mathbf{I}$, denote the simplex vertices, and let h be as in (5.47b). We will denote by $x_c \in \mathbb{R}^n$ the *centroid* of the simplex, defined as

$$x_c \triangleq \frac{1}{n} \sum_{\substack{i=1 \\ i \neq h}}^{n+1} x_i \tag{5.47c}$$

Next, we introduce the three main operations.

Reflection Let $h \in \mathbf{I}$ be as in (5.47b) and let x_c be as in (5.47c). The reflection of $x_h \in \mathbb{R}^n$ to a point denoted as $x^* \in \mathbb{R}^n$ is defined as

$$x^* \triangleq (1+\alpha)\, x_c - \alpha\, x_h, \tag{5.48a}$$

where $\alpha \in \mathbb{R}$, with $\alpha > 0$, is called the *reflection coefficient*.

Expansion of the simplex Let $x^* \in \mathbb{R}^n$ be as in (5.48a) and x_c be as in (5.47c). The expansion of $x^* \in \mathbb{R}^n$ to a point denoted as $x^{**} \in \mathbb{R}^n$ is defined as

$$x^{**} \triangleq \gamma\, x^* + (1-\gamma)\, x_c, \tag{5.48b}$$

where $\gamma \in \mathbb{R}$, with $\gamma > 1$, is called the *expansion coefficient*.

Contraction of the simplex Let $h \in \mathbf{I}$ be as in (5.47b) and x_c be as in (5.47c). The contraction of $x_h \in \mathbb{R}^n$ to a point denoted as $x^{**} \in \mathbb{R}^n$ is defined as

$$x^{**} \triangleq \beta\, x_h + (1-\beta)\, x_c, \tag{5.48c}$$

where $\beta \in \mathbb{R}$, with $0 < \beta < 1$, is called the *contraction coefficient*.

5.7.2 Basic Algorithm

In this section, we describe the basic Nelder and Mead algorithm [NM65]. The extension of O'Neill and the modified restart criterion are discussed later. The algorithm is as follows:

1. Initialization: Given an initial iterate $x_1 \in \mathbb{R}^n$, a scalar c, with $c = 1$ in the initialization, a vector $s \in \mathbb{R}^n$ with user-specified step sizes for each independent parameter, and the set of unit coordinate vectors $\{e_i\}_{i=1}^n$, construct an initial simplex with vertices, for $i \in \{1, \ldots, n\}$,

$$x_{i+1} = x_1 + c\, s^i\, e_i. \tag{5.49}$$

Compute $f(x_i)$, for $i \in \mathbf{I}$.

2. Reflection: Reflect the worst point, that is, compute x^* as in (5.48a).

3. Test whether we got the best point: If $f(x^*) < f(x_l)$, expand the simplex using (5.48b) since further improvement in this direction is likely. If $f(x^{**}) < f(x_l)$, then x_h is replaced by x^{**}, otherwise x_h is replaced by x^*, and the procedure is restarted from 2.

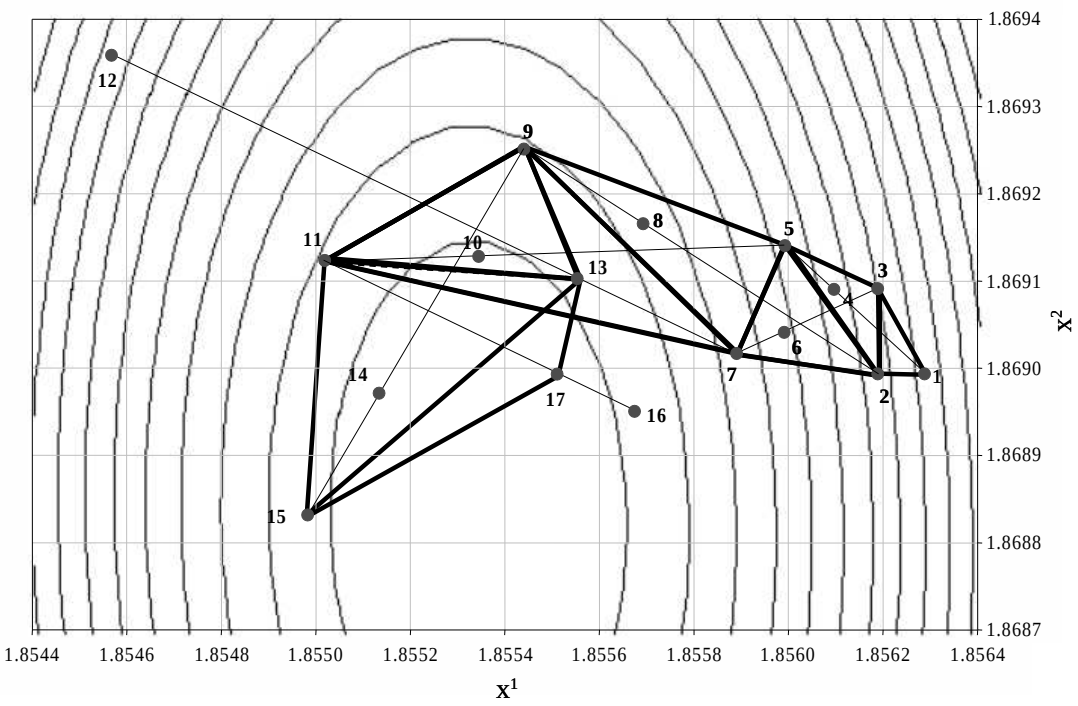


Figure 5.9: *Sequence of iterates generated by the Simplex algorithm.*

4. If it turned out under 3 that $f(x^*) \geq f(x_l)$, then we check if the new point x^* is the worst of all points: If $f(x^*) > f(x_i)$, for all $i \in \mathbf{I}$, with $i \neq h$, we contract the simplex (see 5); otherwise we replace x_h by x^* and go to 2.

5. For the contraction, we first check if we should try a partial outside contraction or a partial inside contraction: If $f(x^*) \geq f(x_h)$, then we try a partial inside contraction. To do so, we leave our indices as is and apply (5.48c). Otherwise, we try a partial outside contraction. This is done by replacing x_h by x^* and applying (5.48c). After the partial inside or the partial outside contraction, we continue at 6.

6. If $f(x^{**}) \geq f(x_h)$[3], we do a total contraction of the simplex by replacing $x_i \leftarrow (x_i + x_l)/2$, for all $i \in \mathbf{I}$. Otherwise, we replace x_h by x^{**}. In both cases, we continue from 2.

[3]Nelder and Mead [NM65] use the strict inequality $f(x^{**}) > f(x_h)$. However, if the user writes the cost function value with only a few representative digits to a text file, then the function looks like a step function if slow convergence is achieved. In such cases, $f(x^{**})$ might sometimes be equal to $f(x_h)$. Experimentally, it has been shown advantageous to perform a total contraction rather than continuing with a reflection. Therefore, the strict inequality has been changed to a weak inequality.

Fig. 5.9 shows a contour plot of a cost function $f\colon \mathbb{R}^n \to \mathbb{R}$ with a sequence of iterates generated by the Simplex algorithm. The sequence starts with constructing an initial simplex x_1, x_2, x_3. x_1 has the highest function value and is therefore reflected, which generates x_4. x_4 is the best point in the set $\{x_1, x_2, x_3, x_4\}$. Thus, it is further expanded, which generates x_5. x_2, x_3 and x_5 now span the new simplex. In this simplex, x_3 is the vertex with the highest function value and hence goes over to x_6 and further to x_7. The process of reflection and expansion is continued again two times, which leads to the simplex spanned by x_7, x_9 and x_{11}. x_7 goes over to x_{12} which turns out to be the worst point. Hence, we do a partial inside contraction, which generates x_{13}. x_{13} is better than x_7 so we use the simplex spanned by x_9, x_{11} and x_{13} for the next reflection. The last steps of the optimization are for clarity not shown.

5.7.3 Stopping Criteria

The first criterion is a test of the variance of the function values at the vertices of the simplex

$$\frac{1}{n}\left(\sum_{i=1}^{n+1}\big(f(x_i)\big)^2 - \frac{1}{n+1}\left(\sum_{i=1}^{n+1} f(x_i)\right)^2\right) < \epsilon^2, \tag{5.50}$$

then the original implementation of the algorithm stops. Nelder and Mead have chosen this stopping criterion based on the statistical problem of finding the minimum of a sum of squares surface. In this problem, the curvature near the minimum yields information about the unknown parameters. A slight curvature indicates a high sampling variance of the estimate. Nelder and Mead argue that in such cases, there is no reason for finding the minimum point with high accuracy. However, if the curvature is marked, then the sampling variance is low and a higher accuracy in determining the optimal parameter set is desirable.

5.7.4 O'Neill's Modification

O'Neill modified the termination criterion by adding a further condition [O'N71]. He checks whether any orthogonal step, each starting from the best vertex of the current simplex, leads to a further improvement of the cost function. He therefore sets $c = 0.001$ and tests if

$$f(x_l) < f(x) \tag{5.51a}$$

for all x defined by

$$x \triangleq x_l + c\, s^i\, e_i, \qquad i \in \{1, \ldots, n\}, \tag{5.51b}$$

where x_l denotes the best known point, and s^i and e_i are as in (5.49).

5.7.5 Modification of Stopping Criteria

In GenOpt, (5.51) has been modified. It has been observed that users sometimes write the cost function value with only few representative digits to the output file. In such cases, (5.51a) is not satisfied if the write statement in the simulation program truncates digits so that the difference $f(x_l) - f(x)$, where $f(\cdot)$ denotes the value that is read from the simulation output file, is zero. To overcome this numerical problem, (5.51b) has been modified to

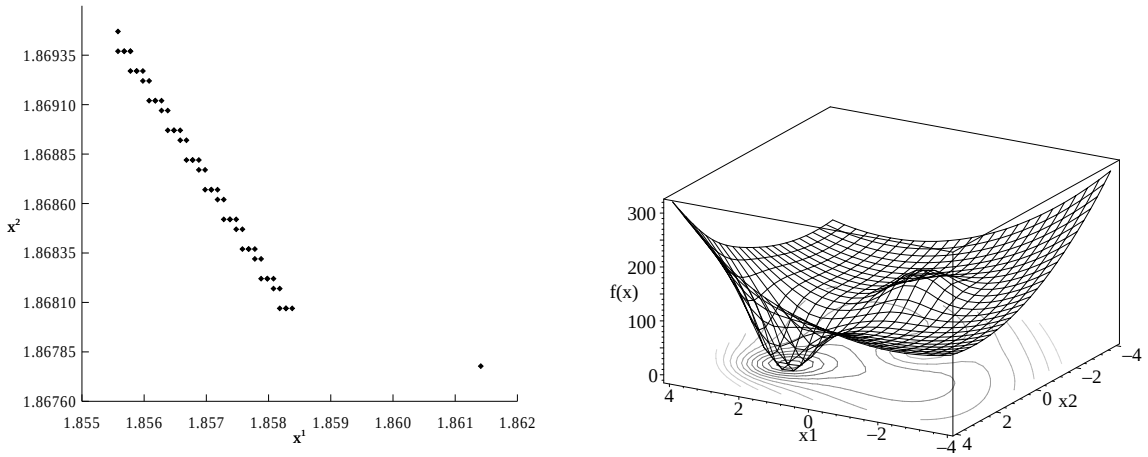
$$x = x_l + \exp(j)\, c\, s^i\, e_i, \qquad i \in \{1, \ldots, n\} \tag{5.51c}$$

where for each direction $i \in \{1, \ldots, n\}$, the counter $j \in \mathbb{N}$ is set to zero for the first trial and increased by one as long as $f(x_l) = f(x)$.

If (5.51a) fails for any direction, then x computed by (5.51c) is the new starting point and a new simplex with side lengths $(c\, s^i)$, $i \in \{1, \ldots, n\}$, is constructed. The point x that failed (5.51a) is then used as the initial point x_l in (5.49).

Numerical experiments showed that during slow convergence the algorithm was restarted too frequently.

Fig. 5.10(a) shows a sequence of iterates where the algorithm was restarted too frequently. The iterates in the figure are part of the iteration sequence near the minimum of the test function shown in Fig. 5.10(b). The algorithm gets close to the minimum with appropriately large steps. The last of these steps can be seen at the right of the figure. After this step, the stopping criterion (5.50) was satisfied which led to a restart check, followed by a new construction of the simplex. From there on, the convergence was very slow due to the small step size. After each step, the stopping criterion was satisfied again which led to a new test of the optimality condition (5.51a), followed by a reconstruction of the simplex. This check is very costly in terms of function evaluations and, furthermore, the restart with a new simplex does not allow increasing the step size, though we are heading locally in the right direction.



(a) Sequence of iterates in the neighborhood of the minimum.

(b) 2-dimensional test function "2D1".

Figure 5.10: *Nelder Mead trajectory.*

O'Neill's modification prevents both excessive checking of the optimality condition as well as excessive reconstruction of the initial simplex. This is done by checking for convergence only after a predetermined number of steps (e.g., after five iterations). However, the performance of the algorithm depends strongly on this number. As an extreme case, a few test runs were done where convergence was checked after each step as in Fig. 5.10(a). It turned out that in some cases no convergence was reached within a moderate number of function evaluations if ϵ in (5.50) is chosen too large, e.g., $\epsilon = 10^{-3}$ (see Tab. 5.1).

To make the algorithm more robust, it is modified based on the following arguments:

1. If the simplex is moving in the same direction in the last two steps, then the search is not interrupted by checking for optimality since we are making steady progress in the moving direction.
2. If we do *not* have a partial inside or total contraction immediately beyond us, then it is likely that the minimum lies in the direction currently being explored. Hence, we do not interrupt the search with a restart.

These considerations have led to two criteria that both have to be satisfied to permit the convergence check according to (5.50), which might be followed by a check for optimality.

First, it is checked if we have done a partial inside contraction or a total contraction. If so, we check if the direction of the latest two steps in which

the simplex is moving has changed by an angle of at least $(\pi/2)$. To do so, we introduce the center of the simplex, defined by

$$x_m \triangleq \frac{1}{n+1} \sum_{i=1}^{n+1} x_i, \tag{5.52}$$

where x_i, $i \in \{1, \ldots, n\}$, are the simplex vertices. We also introduce the normalized direction of the simplex between two steps,

$$d_k \triangleq \frac{x_{m,k} - x_{m,k-1}}{\|x_{m,k} - x_{m,k-1}\|}, \tag{5.53}$$

where $k \in \mathbb{N}$ is the current iteration number.

We determine how much the simplex has changed its direction d_k between two steps by computing the inner product $\langle d_{k-1}, d_k \rangle$. The inner product is equal to the cosine of the angle d_{k-1} and d_k. If

$$\cos \phi_k = \langle d_{k-1},\, d_k \rangle \leq 0, \tag{5.54}$$

then the moving direction of the simplex has changed by at least $\pi/2$. Hence, the simplex has changed the exploration direction. Therefore, a minimum might be achieved and we need to test the variance of the vertices (5.50), possibly followed by a test of (5.51a).

Besides the above modification, a further modification was tested: In some cases, a reconstruction of the simplex after a failed check (5.51a) yields to slow convergence. Therefore, the algorithm was modified so that it continues at point 2 on page 56 without reconstructing the simplex after failing the test (5.51a). However, reconstructing the simplex led in most of the benchmark tests to faster convergence. Therefore, this modification is no longer used in the algorithm.

5.7.6 Benchmark Tests

Tab. 5.1 shows the number of function evaluations and Fig. 5.11 shows the relative number of function evaluations compared to the original implementation for several test cases. The different functions and the parameter settings are given in the Appendix. The only numerical parameter that was changed for the different optimizations is the accuracy, ϵ.

It turned out that modifying the stopping criterion is effective in most cases, particularly if a new simplex is constructed after the check (5.51a) failed. Therefore, the following two versions of the simplex algorithm are implemented in GenOpt:

1. The base algorithm of Nelder and Mead, including the extension of O'Neill. After failing (5.51a), the simplex is *always* reconstructed with the new step size.

	Accuracy							
	$\epsilon = 10^{-3}$				$\epsilon = 10^{-5}$			
Test function	Rosen-brock	2D1	Quad with I ma-trix	Quad with Q ma-trix	Rosen-brock	2D1	Quad with I ma-trix	Quad with Q ma-trix
Original, with recon-struction	137	120	3061	1075	139	109	1066	1165
Original, no recon-struction	136	110	1436	1356	139	109	1433	1253
Modified, with recon-struction	145	112	1296	1015	152	111	1060	1185
Modified, no recon-struction	155	120	1371	1347	152	109	1359	1312

Table 5.1: Comparison of the number of function evaluations for different implementations of the simplex algorithm. See Appendix for the definition of the function.

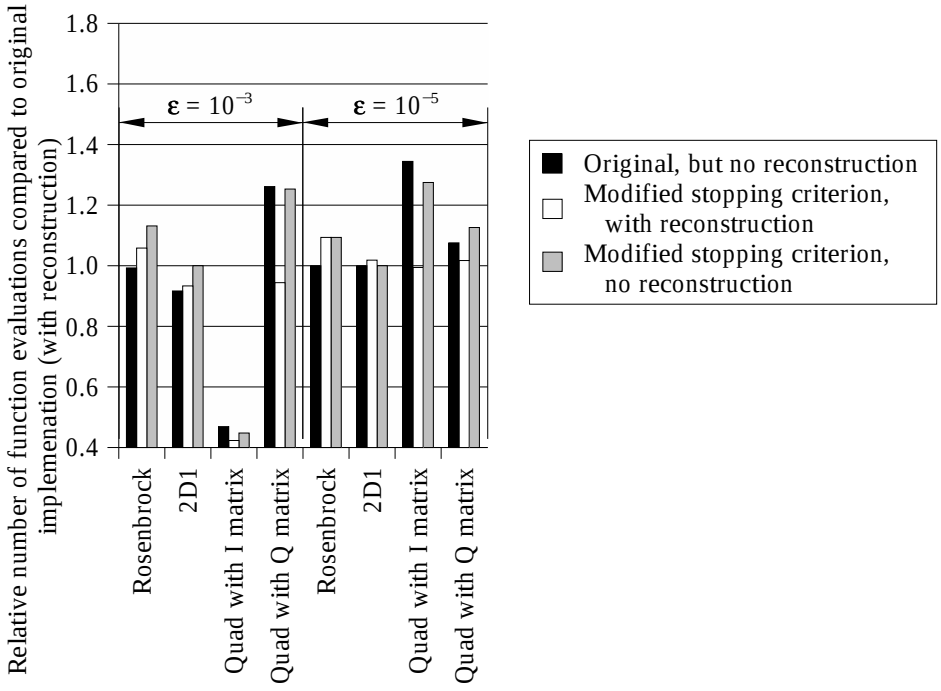


Figure 5.11: *Comparison of the benchmark tests.*

2. The base algorithm of Nelder and Mead, including the extension of O'Neill, but with the modified stopping criterion as explained above. That is, the simplex is only reconstructed if its moving direction changed, and if we have an inside or total construction beyond us.

5.7.7 Keywords

For the Simplex algorithm, the command file (see page 94) must only contain continuous parameters.

To invoke the Simplex algorithm, the `Algorithm` section of the GenOpt command file must have following form:

```
Algorithm{
   Main                    = NelderMeadONeill;
   Accuracy                = Double;   // 0 <  Accuracy
   StepSizeFactor          = Double;   // 0 <  StepSizeFactor
   BlockRestartCheck       = Integer;  // 0 <= BlockRestartCheck
   ModifyStoppingCriterion = Boolean;
}
```

The key words have following meaning:

`Main` The name of the main algorithm.

`Accuracy` The accuracy that has to be reached before the optimality condition is checked. `Accuracy` is defined as equal to ϵ of (5.50), page 58.

`StepSizeFactor` A factor that multiplies the step size of each parameter for (a) testing the optimality condition and (b) reconstructing the simplex. `StepSizeFactor` is equal to c in (5.49) and (5.51c).

`BlockRestartCheck` Number that indicates for how many main iterations the restart criterion is not checked. If zero, restart might be checked after each main iteration.

`ModifyStoppingCriterion` Flag indicating whether the stopping criterion should be modified. If `true`, then the optimality check (5.50) is done only if both of the following conditions are satisfied: (a) in the last step, either a partial inside contraction or total contraction was done, and (b) the moving direction of the simplex has changed by an angle ϕ_k of at least $(\pi/2)$, where ϕ_k is computed using (5.54).

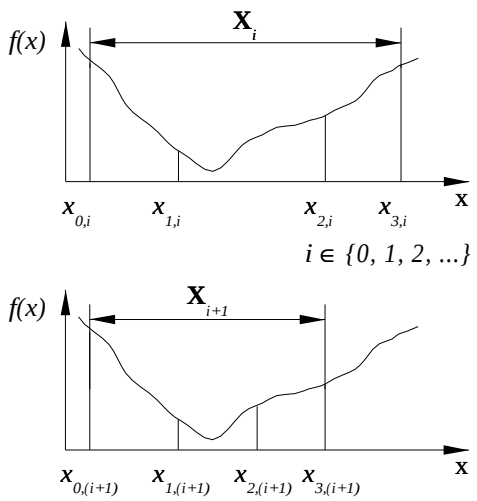


Figure 6.1: Interval division.

6 Algorithms for One-Dimensional Optimization

6.1 Interval Division Algorithms

Interval division algorithm can be used to minimize a function $f : \mathbb{R} \to \mathbb{R}$, (i.e., the function depends on one independent parameter only,) over a user-specified interval. The algorithms do not require derivatives and they require only one function evaluation per interval division, except for the initialization.

First, we explain a master algorithm for the interval division algorithms. The master algorithm is used to implement two commonly used interval division algorithms: The Golden Section search and the Fibonacci Division.

6.1.1 General Interval Division

We now describe the ideas behind the interval division methods. For given $x_0, x_3 \in \mathbb{R}$, with $x_0 < x_3$, let $\mathbf{X} \triangleq [x_0, x_3]$. Suppose we want to minimize $f(\cdot)$ on $\mathbf{X}$, and suppose that $f : \mathbb{R} \to \mathbb{R}$ has a unique minimizer $x^* \in \mathbf{X}$. For some $s \in (0, 1)$, let

$$x_1 \triangleq x_0 + s\,(x_3 - x_0), \tag{6.1}$$

$$x_2 \triangleq x_1 + s\,(x_3 - x_1). \tag{6.2}$$

If $f(x_1) \leq f(x_2)$, then $x^* \in [x_0, x_2]$. Hence, we can eliminate the interval $(x_2, x_3]$ and restrict our search to $[x_0, x_2]$. Similarly, if $f(x_1) > f(x_2)$, then $x^* \in [x_1, x_3]$ and we can eliminate $[x_0, x_1)$. Thus, we reduced the initial inter-

val to a new interval that contains the minimizer, x^*.

Let $i \in \mathbb{N}$ be the iteration number. We want to nest the sequence of intervals

$$[x_{0,(i+1)},\, x_{3,(i+1)}] \subset [x_{0,i},\, x_{3,i}], \qquad i \in \{0,\, 1,\, 2,\, \ldots\}, \tag{6.3}$$

such that we have to evaluate $f(\cdot)$ in each step at one new point only. To do so, we assign the new bounds of the interval such that either $[x_{0,(i+1)},\, x_{3,(i+1)}] = [x_{0,i},\, x_{2,i}]$, or $[x_{0,(i+1)},\, x_{3,(i+1)}] = [x_{1,i},\, x_{3,i}]$, depending on which interval has to be eliminated. By doing so, we have to evaluate only one new point in the interval. It remains to decide where to locate the new point. The Golden Section and Fibonacci Division differ in this decision.

6.1.2 Golden Section Interval Division

Suppose we have three points $x_0 < x_1 < x_3$ in $\mathbf{X} \subset \mathbb{R}$ such that for some $q \in (0, 1)$, to be determined latter,

$$\frac{|x_0 - x_1|}{|x_0 - x_3|} = q. \tag{6.4}$$

Hence,

$$\frac{|x_1 - x_3|}{|x_0 - x_3|} = 1 - q. \tag{6.5}$$

Suppose that x_2 is located somewhere between x_1 and x_3. Define

$$w \triangleq \frac{|x_1 - x_2|}{|x_0 - x_3|}. \tag{6.6}$$

Depending on which interval is eliminated, the interval in the next iteration step will either be of length $(q + w)\,|x_0 - x_3|$, or $(1 - q)\,|x_0 - x_3|$. We select the location of x_2 such that the two intervals are of the same length. Hence,

$$q + w = 1 - q. \tag{6.7}$$

Now, we determine the fraction q. Since we apply the process of interval division recursively, we know by scale similarity that

$$\frac{w}{1 - q} = q. \tag{6.8}$$

Combining (6.7) and (6.8) leads to

$$q^2 - 3q + 1 = 0, \tag{6.9}$$

with solutions

$$q_{1,2} = \frac{3 \pm \sqrt{5}}{2}. \tag{6.10}$$

Since $q < 1$ by (6.4), the solution of interest is

$$q = \frac{3 - \sqrt{5}}{2} \approx 0.382. \tag{6.11}$$

The fractional distances $q \approx 0.382$ and $1 - q \approx 0.618$ correspond to the so-called *Golden Section*, which gives this algorithm its name.

Note that the interval is reduced in each step by the fraction $1 - q$, i.e., we have *linear convergence*. In the m-th iteration, we have

$$\begin{aligned} |x_{0,\,m} - x_{2,\,m}| &= |x_{1,\,m} - x_{3,\,m}| = |x_{0,\,(m+1)} - x_{3,\,(m+1)}| \\ &= (1-q)^{m+1}\,|x_{0,\,0} - x_{3,\,0}|. \end{aligned} \tag{6.12}$$

Hence, the required number of iterations, m, to reduce the initial interval of uncertainty $|x_{0,\,0} - x_{3,\,0}|$ to at least a fraction r defined as

$$r \triangleq \frac{|x_{0,\,m} - x_{2,\,m}|}{|x_{0,\,0} - x_{3,\,0}|} = \frac{|x_{1,\,m} - x_{3,\,m}|}{|x_{0,\,0} - x_{3,\,0}|} \tag{6.13}$$

is given by

$$m = \frac{\ln r}{\ln(1-q)} - 1. \tag{6.14}$$

6.1.3 Fibonacci Division

Another way to divide an interval such that we need one function evaluation per iteration can be constructed as follows: Given an initial interval $[x_{0,\,i}, x_{3,\,i}]$, $i = 0$, we divide it into three segments symmetrically around its midpoint. Let $d_{1,\,i} < d_{2,\,i} < d_{3,\,i}$ denote the distance of the segment endpoints, measured from $x_{0,\,i}$. Then we have by symmetry $d_{3,\,i} = d_{1,\,i} + d_{2,\,i}$. By the bracket elimination procedure explained above, we know that we are eliminating a segment of length $d_{1,\,i}$. Therefore, our new interval is of length $d_{3,\,(i+1)} = d_{2,\,i}$. By symmetry we also have $d_{3,\,(i+1)} = d_{1,\,(i+1)} + d_{2,\,(i+1)}$. Hence, if we construct our segment length such that $d_{3,\,(i+1)} = d_{1,\,(i+1)} + d_{2,\,(i+1)} = d_{2,\,i}$ we can reuse one known point. Such a construction can be done by using *Fibonacci* numbers, which are defined recursively by

$$F_0 \triangleq F_1 \triangleq 1, \tag{6.15}$$

$$F_i \triangleq F_{i-1} + F_{i-2}, \qquad i \in \{2,\,3,\,\ldots\}. \tag{6.16}$$

The first few numbers of the Fibonacci sequence are $\{1,\,1,\,2,\,3,\,5,\,8,\,13,\,21,\,\ldots\}$. The length of the intervals $d_{1,\,i}$ and $d_{2,\,i}$, respectively, are then given by

$$d_{1,\,i} = \frac{F_{m-i}}{F_{m-i+2}}, \quad d_{2,\,i} = \frac{F_{m-i+1}}{F_{m-i+2}}, \qquad i \in \{0,\,1,\,\ldots\,,\,m\}, \tag{6.17}$$

where $m > 0$ describes how many iterations will be done. Note that m must be known prior to the first interval division. Hence, the algorithm must be

stopped after m iterations.

The reduction of the length of the uncertainty interval per iteration is given by

$$\frac{d_{3,\,(i+1)}}{d_{3,\,i}} = \frac{d_{2,\,i}}{d_{1,\,i} + d_{2,\,i}} = \frac{\frac{F_{m-i+1}}{F_{m-i+2}}}{\frac{F_{m-i}}{F_{m-i+2}} + \frac{F_{m-i+1}}{F_{m-i+2}}} = \frac{F_{m-i+1}}{F_{m-i+2}}. \tag{6.18}$$

After m iterations, we have

$$\begin{aligned} \frac{d_{3,\,m}}{d_{3,\,0}} &= \frac{d_{3,\,m}}{d_{3,\,(m-1)}} \frac{d_{3,\,(m-1)}}{d_{3,\,(m-2)}} \cdots \frac{d_{3,\,2}}{d_{3,\,1}} \frac{d_{3,\,1}}{d_{3,\,0}} \\ &= \frac{F_2}{F_3} \frac{F_3}{F_4} \cdots \frac{F_m}{F_{m+1}} \frac{F_{m+1}}{F_{m+2}} = \frac{2}{F_{m+2}}. \end{aligned} \tag{6.19}$$

The required number of iterations m to reduce the initial interval $d_{3,\,0}$ to at least a fraction r, defined by (6.13), can again be obtained by expansion from

$$\begin{aligned} r &= \frac{d_{2,\,m}}{d_{3,\,0}} = \frac{d_{3,\,(m+1)}}{d_{3,\,0}} = \frac{d_{3,\,(m+1)}}{d_{3,\,m}} \frac{d_{3,\,m}}{d_{3,\,(m-1)}} \cdots \frac{d_{3,\,2}}{d_{3,\,1}} \frac{d_{3,\,1}}{d_{3,\,0}} \\ &= \frac{F_1}{F_2} \frac{F_2}{F_3} \cdots \frac{F_m}{F_{m+1}} \frac{F_{m+1}}{F_{m+2}} = \frac{1}{F_{m+2}}. \end{aligned} \tag{6.20}$$

Hence, m is given by

$$m = \arg\min_{m \in \mathbb{N}} \left\{ m \mid r \geq \frac{1}{F_{m+2}} \right\}. \tag{6.21}$$

6.1.4 Comparison of Efficiency

The Golden Section is more efficient than the Fibonacci Division. Comparing the reduction of the interval of uncertainty, $|x_{0,\,m} - x_{3,\,m}|$, in the limiting case for $m \to \infty$, we obtain

$$\lim_{m \to \infty} \frac{|x_{0,\,m} - x_{3,\,m}|_{GS}}{|x_{0,\,m} - x_{3,\,m}|_{F}} = \lim_{m \to \infty} \frac{F_{m+2}}{2} (1-q)^m = 0.95. \tag{6.22}$$

6.1.5 Master Algorithm for Interval Division

The following master algorithm explains the steps of the interval division algorithm.

Algorithm 6.1.1 (Model Interval Division Algorithm)

Data:	x_0, x_3. Procedure that returns r_i, defined as $r_i \triangleq \lvert x_{0,i} - x_{2,i}\rvert / \lvert x_{0,0} - x_{3,0}\rvert$.
Step 0:	*Initialize* $\Delta x = x_3 - x_0$, $x_2 = x_0 + r_1 \, \Delta x$, $x_1 = x_0 + r_2 \, \Delta x$, $f_1 = f(x_1)$, $f_2 = f(x_2)$, and $i = 2$.
Step 1:	*Iterate.* Replace i by $i + 1$. If $(f_2 < f_1)$ Set $x_0 = x_1$, $x_1 = x_2$, $f_1 = f_2$, $x_2 = x_3 - r_i \, \Delta x$, and $f_2 = f(x_2)$. else Set $x_3 = x_2$, $x_2 = x_1$, $f_2 = f_1$, $x_1 = x_0 + r_i \, \Delta x$, $f_1 = f(x_1)$.
Step 2:	Stop or go to Step 1.

6.1.6 Keywords

For the Golden Section and the Fibonacci Division algorithm, the command file (see page 94) must contain only one continuous parameter.

To invoke the Golden Section or the Fibonacci Division algorithm, the `Algorithm` Section of the GenOpt command file must have following form:

```
Algorithm{
   Main              = GoldenSection | Fibonacci;
  [AbsDiffFunction   = Double;  |   // 0 < AbsDiffFunction
   IntervalReduction = Double;  ]   // 0 < IntervalReduction
}
```

The keywords have following meaning

`Main` The name of the main algorithm.

The following two keywords are optional. If none of them is specified, then the algorithm stops after `MaxIte` function evaluations (i.e., after `MaxIte`-2 iterations), where `MaxIte` is specified in the section `OptimizationSettings`. If both of them are specified, an error occurs.

`AbsDiffFunction` The absolute difference defined as

$$\Delta f \triangleq |\min\{f(x_0),\, f(x_3)\} - \min\{f(x_1),\, f(x_2)\}|. \tag{6.23}$$

If Δf is lower than `AbsDiffFunction`, the search stops successfully.
Note: Since the maximum number of interval reductions must be known for the initialization of the Fibonacci algorithm, this keyword can be used only for the Golden Section algorithm. It must not be specified for the Fibonacci algorithm.

`IntervalReduction` The required maximum fraction, r, of the end interval length relative to the initial interval length (see equation (6.13)).

7 Algorithms for Parametric Runs

7.1 Parametric Runs by Single Variation

7.1.1 Algorithm Description

The `Parametric` algorithm allows doing parametric runs where one parameter at a time is varied and all other parameters are fixed at their initial values (specified by the keyword `Ini`). Each parameter must have a lower and upper bound. For the logarithmic scale, lower and upper bound must be bigger than zero. To allow negative increments, the lower bound can be larger than the upper bound. The absolute value of the keyword `Step` defines in how many intervals each coordinate axis will be divided. If `Step` < 0, then the spacing is logarithmic, otherwise it is linear. Set `Step` $= 0$ to keep the parameter always fixed at the value specified by `Ini`.

Next, we explain how the spacing is computed. For simplicity, the explanation is done for one parameter. Let $l \triangleq$ `Min`, $u \triangleq$ `Max` and $m \triangleq |$`Step`$|$, where `Min`, `Max` and `Step` are specified in the command file.

If `Step` < 0, we compute, for $i \in \{0, \ldots, m\}$,

$$p = \frac{1}{m} \log \frac{u}{l}, \tag{7.1a}$$

$$x_i = l\, 10^{p\, i}. \tag{7.1b}$$

If `Step` > 0, we compute, for $i \in \{0, \ldots, m\}$,

$$x_i = l + \frac{i}{m} (u - l). \tag{7.1c}$$

Example 7.1.1 (Parametric run with logarithmic and linear spacing)
Suppose the parameter specification is of the form

```
Vary{
   Parameter{ Name = x1; Ini = 5; Step = -2; Min = 10; Max = 1000; }
   Parameter{ Name = x2; Ini = 3; Step = 1;  Min = 2;  Max = 20;   }
}
```

and the cost function takes two arguments, $x_1, x_2 \in \mathbb{R}$. Then, the cost function will be evaluated at the points
$(x_1, x_2) \in \{(10, 3),\ (100, 3),\ (1000, 3),\ (5, 2),\ (5, 20)\}$. □

7.1.2 Keywords

For this algorithm, the command file (see page 94) can contain both, continuous and discrete parameters.

The `Parametric` algorithm is invoked by the following specification in the command file:

```
Algorithm{
   Main = Parametric;
   StopAtError = true | false;
}
```

The keywords have following meaning:

`Main` The name of the main algorithm.

`StopAtError` If `true`, then the parametric run stops if a simulation error occurs. If `false`, then the parametric run does not stop if a simulation error occurs. The failed function evaluation will be assigned the function value zero. For information, an error message will be written to the user interface and the optimization log file.

Note that the whole section `OptimizationSettings` of the command file is ignored.

7.2 Parametric Runs on a Mesh

7.2.1 Algorithm Description

The `EquMesh` algorithm allows making parametric runs on an orthogonal, equidistant grid that is spanned in the space of the independent parameters. To do so, each independent parameter must have a lower and upper bound. The value of `Step` (which must be an integer greater than or equal to zero) specifies into how many intervals each axis will be divided. The spacing is computed according to (7.1c). Hence, we also allow `Min` > `Max`.

Example 7.2.1 (Parametric run on a mesh)
Suppose the parameter specification is of the form

```
Vary{
   Parameter{ Name = x0; Min = -10; Ini = 99; Max = 10; Step = 1; }
   Parameter{ Name = x1; Min =   1; Ini = 99; Max = -1; Step = 2; }
}
```

and the cost function takes two arguments, $x_1, x_2 \in \mathbb{R}$. Then, the cost function will be evaluated at the points
$(x_1, x_2) \in \triangleq \{(-10, 1), (10, 1), (-10, 0), (10, 0), (-10, -1), (10, -1)\}$. □

If the value of `Step` is equal to zero, then this parameter is fixed at the value specified by `Min`.

Note that the number of function evaluations increases exponentially with the number of independent parameters. For example, a 5-dimensional grid with 2 intervals in each dimension requires $3^5 = 243$ function evaluations, whereas a 10-dimensional grid would require $3^{10} = 59049$ function evaluations.

7.2.2 Keywords

For this algorithm, the command file (see page 94) can contain both, continuous and discrete parameters.

The `EquMesh` algorithm is invoked by the following specification in the command file:

```
Algorithm{
   Main        = EquMesh;
   StopAtError = true | false;
}
```

The keywords have following meaning:

`Main` The name of the main algorithm.

`StopAtError` If `true`, then the parametric run stops if a simulation error occurs. If `false`, then the parametric run does not stop if a simulation error occurs. The failed function evaluation will be assigned the function value zero. For information, an error message will be written to the user interface and the optimization log file.

Note that the whole section `OptimizationSettings` of the command file is ignored.

8 Constraints

For some optimization problems it is necessary to impose constraints on the independent variables and/or the dependent variables, as the following example shows.

Example 8.0.2 Suppose we want to minimize the heating energy of a building, and suppose that the mass flow $\dot{m}$ of the heating system is an independent variable, with constraints $0 \leq \dot{m} \leq 1$. Without using constraints, the minimum energy consumption would be achieved for $\dot{m} = 0$, since then the heating system is switched off. To solve this problem, we can impose a constraint on a dependent variable. One possibility is to add a "penalty" term to the energy consumption. This could be such that every time a thermal comfort criterion (which is a dependent variable) is violated, a large positive number is added to the energy consumption. Thus, if $\mathrm{ppd}(x)$, with $\mathrm{ppd}\colon \mathbb{R}^n \to \mathbb{R}$, denotes the predicted percent of dissatisfied people (in percentage), and if we require that $\mathrm{ppd}(x) \leq 10\%$, we could use the inequality constraint $g(x) \triangleq \mathrm{ppd}(x) - 10 \leq 0$.

□

In Section 8.1.1, the method that is used in GenOpt to implement box constraints is described. In Section 8.2, penalty and barrier methods that can be used to implement constraints on dependent variables are described. They involve reformulating the cost function and, hence, are problem specific and have to be implemented by the user.

8.1 Constraints on Independent Variables

8.1.1 Box Constraints

Box constraints are constant inequality constraints that define a feasible set as

$$\mathbf{X} \triangleq \left\{ x \in \mathbb{R}^n \mid l^i \leq x^i \leq u^i,\ i \in \{1, \ldots, n\} \right\}, \tag{8.1}$$

where $-\infty \leq l^i < u^i \leq \infty$ for $i \in \{1, \ldots, n\}$.

In GenOpt, box constraints are either implemented directly in the optimization algorithm by setting $f(x) = \infty$ for unfeasible iterates, or, for some algorithms, the independent variable $x \in \mathbf{X}$ is transformed to a new unconstrained variable which we will denote in this section by $t \in \mathbb{R}^n$.

Instead of optimizing the constrained variable $x \in \mathbf{X}$, we optimize with respect to the unconstrained variable $t \in \mathbb{R}^n$. The transformation ensures that all variables stay feasible during the iteration process. In GenOpt, the following transformations are used:

If $l^i \leq x^i$, for some $i \in \{1, \ldots, n\}$,

$$t^i = \sqrt{x^i - l^i}, \tag{8.2a}$$

$$x^i = l^i + (t^i)^2. \tag{8.2b}$$

If $l^i \leq x^i \leq u^i$, for some $i \in \{1, \ldots, n\}$,

$$t^i = \arcsin\left(\sqrt{\frac{x^i - l^i}{u^i - l^i}}\right), \tag{8.2c}$$

$$x^i = l^i + (u^i - l^i)\,\sin^2 t^i. \tag{8.2d}$$

If $x^i \leq u^i$, for some $i \in \{1, \ldots, n\}$,

$$t^i = \sqrt{u^i - x^i}, \tag{8.2e}$$

$$x^i = u^i - (t^i)^2. \tag{8.2f}$$

8.1.2 Coupled Linear Constraints

In some cases the constraints have to be formulated in terms of a linear system of equations of the form

$$A\,x = b, \tag{8.3}$$

where $A \in \mathbb{R}^m \times \mathbb{R}^n$, $x \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, and $\text{rank}(A) = m$.

There are various algorithms that take this kind of restriction into account. However, such restrictions are rare in building simulation and thus not implemented in GenOpt. If there is a need to impose such restrictions, they can be included by adding an appropriate optimization algorithm and retrieving the coefficients by using the methods offered in GenOpt's class `Optimizer`.

8.2 Constraints on Dependent Variables

We now discuss the situation where the constraints are non-linear and defined by

$$g(x) \leq 0, \tag{8.4}$$

where $g\colon \mathbb{R}^n \to \mathbb{R}^m$ is once continuously differentiable. (8.4) also allows formulating equality constraints of the form

$$h(x) = 0, \tag{8.5}$$

for $h\colon \mathbb{R}^n \to \mathbb{R}^m$, which can be implemented by using penalty functions. In example, one can define $g^i(x) \triangleq h^i(x)^2$ for $i \in \{1, \ldots, m\}$. Then, since $g^i(\cdot)$ is non-negative, the only feasible value is $g(\cdot) = 0$. Thus, we will only discuss the case of inequality constraints of the form (8.4).

Such a constraint can be taken into account by adding *penalty* or *barrier* functions to the cost function, which are multiplied by a positive weighting factor μ that is monotonically increased (for penalty functions) or monotonically decreased to zero (for barrier functions).

We now discuss the implementation of barrier and penalty functions.

8.2.1 Barrier Functions

Barrier functions impose a punishment if the dependent variable gets close to the boundary of the feasible region. The closer the variable is to the boundary, the higher the value of the barrier function becomes.
To implement a barrier function for $g(x) \leq 0$, where $g\colon \mathbb{R}^n \to \mathbb{R}^m$ is a continuously differentiable function whose elements are strictly monotone increasing, the cost function $f\colon \mathbb{R}^n \to \mathbb{R}$ can be modified to

$$\widetilde{f}(x,\mu) \triangleq f(x) + \mu \frac{1}{\sum_{i=1}^{m} g^i(x)} \tag{8.6}$$

where $\widetilde{f}\colon \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$. The optimization algorithm is then applied to the new function $\widetilde{f}(x,\mu)$. Note that (8.6) requires that x is in the interior of the feasible set[1].

A drawback of barrier functions is that the boundary of the feasible set can not be reached. By selecting the weighting factors small, one can get close to the boundary. However, too small a weighting factor can cause the cost function to be ill-conditioned, which can cause problems for the optimization algorithm.

Moreover, if the variation of the iterates between successive iterations is too big, then the feasible boundary can be crossed. Such a behavior must be prevented by the optimization algorithm, which can produce additional problems.

For barrier functions, one can start with a moderately large weighting factor μ and let μ tend to zero during the optimization process. That is, one constructs a sequence

$$\mu_0 > \ldots > \mu_i > \mu_{i+1} > \ldots > 0. \tag{8.7}$$

Section 8.2.3 shows how μ_i can be computed in the coarse of the optimization.

Barrier functions do not allow formulating equality constraints of the form (8.5).

8.2.2 Penalty Functions

In contrast to barrier functions, penalty functions allow crossing the boundary of the feasible set, and they allow implementation of equality constraints of the form (8.5). Penalty functions add a positive term to the cost function if a constraint is violated.
To implement a penalty function for $g(x) \leq 0$, where $g\colon \mathbb{R}^n \to \mathbb{R}^m$ is once continuously differentiable and each element is strictly monotone decreasing, the cost function $f\colon \mathbb{R}^n \to \mathbb{R}$ can be modified to

$$\widetilde{f}(x,\mu) \triangleq f(x) + \mu \sum_{i=1}^{m} \max(0, g^i(x))^2, \tag{8.8}$$

[1] I.e., x satisfies the strict inequality $g(x) > 0$.

where $\widetilde{f}\colon \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$ is once continuously differentiable in x. The optimization algorithm is then applied to the new function $\widetilde{f}(x, \mu)$.

As for the barrier method, selecting the weighting factor μ is not trivial. Too small a value for μ produces too big a violation of the constraint. Hence, the boundary of the feasible set can be exceeded by an unacceptable amount. Too large a value of μ can lead to ill-conditioning of the cost function, which can cause numerical problems.

The weighting factors have to satisfy

$$0 < \mu_0 < \ldots < \mu_i < \mu_{i+1} < \ldots, \tag{8.9}$$

with $\mu_i \to \infty$, as $i \to \infty$. See Section 8.2.3 for how to adjust μ_i.

8.2.3 Implementation of Barrier and Penalty Functions

We now discuss how the weighting factors μ_i can be adjusted. For $i \in \mathbb{N}$, let $x^*(\mu_i)$ be defined as the solution

$$x^*(\mu_i) \triangleq \arg\min_{x \in \mathbf{X}} \widetilde{f}(x, \mu_i), \tag{8.10}$$

where $\widetilde{f}(x, \mu_i)$ is as in (8.6) or (8.8), respectively. Then, we initialize $i = 0$, select an initial value $\mu_0 > 0$ and compute $x^*(\mu_0)$. Next, we select a μ_{i+1} such that it satisfies (8.7) (for barrier functions) or (8.9) (for penalty functions), and compute $x^*(\mu_{i+1})$, using the initial iterate $x^*(\mu_i)$, and increase the counter i to $i + 1$. This procedure is repeated until μ_i is sufficiently close to zero (for barrier functions) or sufficiently large (for penalty functions).

To recompute the weighting factors μ_i, users can request GenOpt to write a counter to the simulation input file, and then compute μ_i as a function of this counter. The value of this counter can be retrieved by setting the keyword `WriteStepNumber` in the optimization command file to `true`, and specifying the string `%stepNumber%` in the simulation input template file. GenOpt will replace the string `%stepNumber%` with the current counter value when it writes the simulation input file. The counter starts with the value 1 and its increment is 1.

Users who implement their own optimization algorithm in GenOpt can call the method `increaseStepNumber(...)` in the class `Optimizer` to increase the counter. If the keyword `WriteStepNumber` in the optimization command file is set to `true`, the method calls the simulation to evaluate the cost function for the new value of this counter. If `WriteStepNumber` is `false`, no new function evaluation is performed by this method since the cost function does not depend on this counter.

Input Files

initialization:	Specification of file location (input files, output files, log files, etc.)
command:	Specification of parameter names, initial values, upper/lower bounds, optimization algorithm, etc.
configuration:	Configuration of simulation program (error indicators, start command, etc.)
simulation input template:	Templates of simulation input files

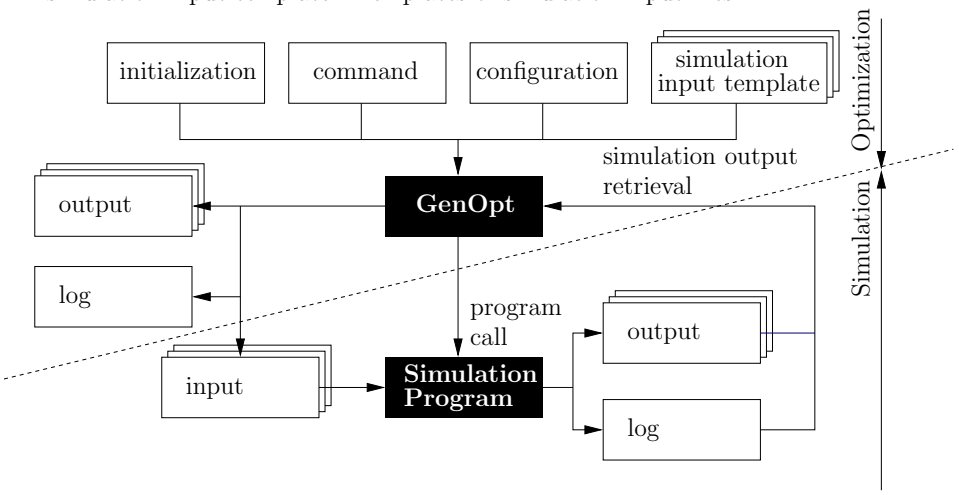


Figure 9.1: *Interface between GenOpt and the simulation program that calculates the cost function.*

9 Program

GenOpt is divided into a kernel part and an optimization part. The kernel reads the input files, calls the simulation program, stores the results, writes output files, etc. The optimization part contains the optimization algorithms. It also contains classes of mathematical functions such as those used in linear algebra.

Since there is a variety of simulation programs and optimization algorithms, GenOpt has a simulation program interface and an optimization algorithm interface. The simulation program interface allows using any simulation software to evaluate the cost function (see below for the requirements on the simulation program), and allows implementing new optimization algorithms with little effort.

9.1 Interface to the Simulation Program

Text files are used to exchange data with the simulation program and to specify how to start the simulation program. This makes it possible to couple

any simulation program to GenOpt without requiring code adaptation on either the GenOpt side or the simulation program side. The simulation program must satisfy the following requirements:

1. The simulation program must read its input from one or more text files, must write the value of the cost function to a text file, and must write error messages to a text file.
2. It must be able to start the simulation program by a command and the simulation program must terminate automatically. This means that the user does not have to open the input file manually and shut down the simulation program once the simulation is finished.

The simulation program may be a commercially available program or one written by the user.

9.2 Interface to the Optimization Algorithm

The large variety of optimization algorithms led to the development of an open interface that allows easy implementation of optimization algorithms. Users can implement their own algorithms and add them to the library of available optimization algorithms without having to adapt and recompile GenOpt. To implement a new optimization algorithm, the optimization algorithm must be written according to the guidelines of Section 9.4. Thus, GenOpt can not only be used to do optimization with built-in algorithms, but it can also be used as a framework for developing, testing and comparing optimization algorithms.

Fig. 9.2 shows GenOpt's program structure. The class `Optimizer` is the superclass of each optimization algorithm. It offers all the functions required for retrieval of parameters that specify the optimization settings, performing the evaluation of the cost function and reporting results. For a listing of its methods, see `http://simulationresearch.lbl.gov` or the Javadoc code documentation that comes with GenOpt's installation.

9.3 Package `genopt.algorithm`

The Java package `genopt.algorithm` consists of all classes that contain mathematical formulas that are used by the optimization algorithm. The following packages belong to `genopt.algorithm`.

`genopt.algorithm` This package contains all optimization algorithms. The abstract class `Optimizer`, which must be inherited by each optimization algorithm, is part of this package.

`genopt.algorithm.util.gps` contains a model Generalized Pattern Search optimization algorithm.

`genopt.algorithm.util.linesearch` contains classes for doing a line search along a given direction.

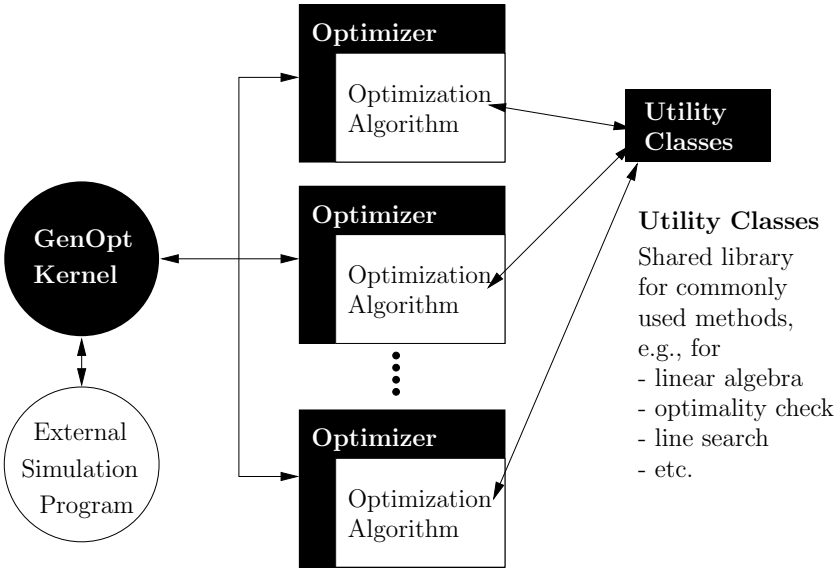


Figure 9.2: Implementation of optimization algorithms into GenOpt.

`genopt.algorithm.util.math` contains classes for mathematical operations.

`genopt.algorithm.util.optimality` contains classes that can be used to check whether a variable value is at a minimum point or not.

`genopt.algorithm.util.pso` contains a model Particle Swarm Optimization algorithm.

These packages are documented in the Javadoc source code documentation that comes with GenOpt's.

9.4 Implementing a New Optimization Algorithm

To implement a new optimization algorithm, you must write a Java class that has the syntax shown in Fig. 9.3. The class must use the methods of the abstract class `Optimizer` to evaluate the cost function and to report the optimization steps. The methods of the `Optimizer` class are documented in the Javadoc source code documentation.

Follow these steps to implement and use your own optimization algorithm:

1. Place the byte-code (*ClassName*`.class`) in the directory `genopt/algorithm` (on Linux or Unix) or `genopt\algorithm` (on Windows).
2. Set the value of the keyword `Main` in the `Algorithm` section of the optimization command file to the name of the optimization class (without file extension).
3. Add any further keywords that the algorithm requires to the `Algorithm` section. The keywords must be located *after* the entry `Main` of the optimization command file. The keywords must be in the same sequence as they are called in the optimization code.
4. Call the method `Optimizer.report(final Point, final boolean)` after evaluating the cost function. Otherwise, the result will not be reported.
5. Call either the method `Optimizer.increaseStepNumber()` or `Optimizer.increaseStepNumber(finalPoint)` after the optimization algorithm converged to some point. These methods increase a counter that can be used to add penalty or barrier functions to the cost function. In particular, the methods `Optimizer.increaseStepNumber()` and `Optimizer.increaseStepNumber(finalPoint)` increase the variable `stepNumber` (see Section 8) by one.

```
package genopt.algorithm;

import genopt.GenOpt;
import genopt.lang.OptimizerException;
import genopt.io.InputFormatException;

public class ClassName extends Optimizer{

   public ClassName (GenOpt genOptData)
      throws InputFormatException, OptimizerException,
             IOException, Exception
   {
      // set the mode to specify whether the
      // default transformations for the box
      // constraints should be used or not
      int constraintMode = xxxx;
      super(genOptData, constraintMode);

      // remaining code of the constructor
   }

   public int run() throws OptimizerException, IOException
   {
      // the code of the optimization algorithm
   }

   // add any further methods and data members
}
```

Figure 9.3: *Code snippet that specifies how to implement an optimization algorithm.*

10 Installing and Running GenOpt

10.1 Installing GenOpt

GenOpt 2.0.0 β is written in Java 2 v1.4.1 to ensure platform independence. To use GenOpt, a Java interpreter for Java 2 v1.4.1 or higher must be installed, such as Sun's Java Runtime Environment (JRE). Users who want to add their own optimization algorithm must also have a Java compiler. Sun's Java Development Kit (JDK) contains a compiler and the runtime environment. GenOpt has been tested with Sun's Java 2 v1.4.1 .

JRE and JDK can be downloaded from `http://java.sun.com/products/`. JRE and JDK also run on the Windows operating system.

After Java has been installed, the file `go_prg.linux` (self extracting file, for Linux only) or `go_prg.zip` must be downloaded from `http://SimulationResearch.lbl.gov` and extracted. On Linux, the file `go_prg.linux` can be extracted by typing

```
chmod +x go_prg.linux
./go_prg.linux
```

On Unix, the file `go_prg.zip` can be extracted by typing

```
unzip go_prg.zip go_prg
```

On Windows, the file `go_prg.zip` can be extracted with the software WinZip.

10.2 System Configuration for JDK Installation

In the instructions below, "`.`" stands for the current directory and the directory `go_prg` contains the directory `genopt` (i.e., `go_prg/genopt`) where the Java class files of GenOpt are stored.

10.2.1 Linux/Unix

The installation is explained for the bash-shell and the C-shell. We assume that the Java binaries are in the directory `/usr/local/jdk/bin`. For the bash-shell, the following lines must be added to the file `~/.bashrc`:

```
PATH="$PATH:/usr/local/jdk/bin"
CLASSPATH="$CLASSPATH:.:$HOME/go_prg"
export PATH CLASSPATH
```

For the C-shell, the following lines must be added to the file `~/.cshrc`:

```
set PATH=PATH:/usr/local/jdk/bin
setenv CLASSPATH CLASSPATH:.:$HOME/go_prg
```

10.2.2 Microsoft Windows

The `PATH` variable of the system must contain the directory where the Java Virtual Machine is located. For example,

```
SET PATH=%PATH%;C:\prog\jdk\bin
```

There must also be a CLASSPATH variable that points to the Java classes. This variable has the form

```
SET CLASSPATH=%CLASSPATH%;.;C:\prog\go_prg
```

In Windows 2000, both variables can be specified under "Start → Settings → Control Panel → System → Advanced → Environment Variables...". In Windows 95 and 98, they can be specified in the `autoexec.bat` file.

10.3 Starting an Optimization with JDK Installation

GenOpt can be run either with a graphical user interface (GUI), or as a console application. The GUI has an online chart that shows the optimization progress of the cost function and of certain parameters. Any values can be added to or removed from the chart during runtime. The console application allows running GenOpt with a batch job for several sequential optimizations or starting GenOpt over a remote connection, e.g., using `telnet`. The version with the GUI can be started with the command

```
java genopt.WinGenOpt
```

and the console version can be started with the command

```
java genopt.GenOpt [Optimization Initialization File]
```

In these commands, `java` is the name of the Java virtual machine (that interprets the byte code), and `genopt.WinGenOpt` (or `genopt.GenOpt`) is the full name of the main class. The brackets indicate that the last parameter is optional. The optimization initialization file can also be specified after starting GenOpt.

10.4 System Configuration for JRE Installation

First, make sure that the path variable points to the path where the `jre` binary (i.e., `jre.exe`) is located. If this is not the case, set the path variable as described in Section 10.2.

We recommend setting an environment variable, similar to the `CLASSPATH` variable, to the `genopt` directory. This can be done in the same way as described for the `CLASSPATH` variable for the JDK installation. Note that on Windows platforms, JRE will ignore the `CLASSPATH` environment variable. For both Windows and Solaris platforms, the `-cp` option is recommended to specify an application's class path (see below).

10.5 Starting an Optimization with JRE Installation

The GUI version can be launched with the command

```
jre -cp %CLASSPATH% genopt.WinGenOpt
```

and the console version with

```
jre -cp %CLASSPATH% genopt.GenOpt [OptInitializationFile]
```

where `CLASSPATH` is the name of the environment variable that points to the `genopt` directory.

11 Setting Up an Optimization Problem

We will now discuss how to set up an optimization problem.

First, define a cost function. The cost function is the function that needs to be minimized. It must be evaluated by an external simulation program that satisfies the requirements listed on page 79. To maximize a cost function, change the sign of the cost function to turn the maximization problem into a minimization problem.

Next, specify possible constraints on the independent variables or on dependent variables (dependent variables are values that are computed in the simulation program). To do so, use the default scheme for box constraints on the independent variables or add penalty or barrier functions to the cost function as described in Chapter 8.

Next, make sure that the simulation program writes the cost function value to the simulation output file. *It is important that the cost function value is written to the output file without truncating any digits* (see Section 11.3). For example, if the cost function is computed by a Fortran program in double precision, it is recommended to use the `E24.16` format in the write statement.

In the simulation output file, the cost function value must be indicated by a string that stands in front of the cost function value (see page 91).

Then, specify the files described in Section 11.1 and, if required, implement pre- and post-processing, as described in Section 11.2.

11.1 File Specification

This section defines the file syntax for GenOpt. The directory `example` of the GenOpt installation contains several examples.

The following notation will be used to explain the syntax:

1. Text that is part of the file is written in `fixed width fonts`.
2. `|` stands for possible entries. Only one of the entries that are separated by `|` is allowed.
3. `[ ]` indicates optional values.
4. The file syntax follows the Java convention. Hence,
   (a) `//` indicates a comment on a single line,
   (b) `/*` and `*/` enclose a comment,
   (c) the equal sign, `=`, assigns values,
   (d) a statement has to be terminated by a semi-colon, `;`,
   (e) curly braces, `{ }`, enclose a whole section of statements, and

(f) the syntax is case sensitive.

The following basic types are used:

`String`	Any sequence of characters. If the sequence contains a blank character, it has to be enclosed in apostrophes ("). If there are apostrophes within quoted text, they must be specified by a leading backslash (i.e., \"). Similarly, a backslash must be preceded by another backslash (i.e., "c:\\go_prg").
`StringReference`	Any name of a variable that appears in the same section.
`Integer`	Any integer value.
`Double`	Any double value (including integer).
`Boolean`	Either `true` or `false`

The syntax of the GenOpt files is structured into sections of parameters that belong to the same object. The sections have the form

```
ObjectKeyWord { Object }
```

where `Object` can either be another `ObjectKeyWord` or an assignment of the form

```
Parameter = Value ;
```

Some variables can be referenced. References have to be written in the form

```
Parameter = ObjectKeyWord1.ObjectKeyWord2.Value ;
```

where `ObjectKeyWord1` refers to the root of the object hierarchy as specified in the corresponding file.

11.1.1 Initialization File

The initialization file specifies

1. where the *files* of the current optimization problems are located,
2. which simulation files the user likes to have saved for latter inspection,
3. what additional strings have to be passed to the command that starts the simulation (such as the name of the simulation input file),
4. what number in the simulation output file is a cost function value,
5. whether and if so, how, the cost function value(s) have to be post-processed, and
6. which simulation program is being used.

The sections must be specified in the order shown below. The order of the keywords in each section is arbitrary, as long as the numbers that follow some keywords (such as `File1`) are in increasing order.

The initialization file syntax is

```
Simulation {
  Files {
    Template {
      File1 = String | StringReference;
    [ Path1 = String | StringReference; ]
      [ File2 = String | StringReference;
      [ Path2 = String | StringReference; ]
        [ ... ] ]
    }
    Input { // the number of input file must be equal to
            // the number of template files
      File1     = String | StringReference;
    [ Path1     = String | StringReference; ]
    [ SavePath1 = String | StringReference; ]
      [ File2     = String | StringReference;
      [ Path2     = String | StringReference; ]
      [ SavePath2 = String | StringReference; ]
        [ ... ] ]
    }
    Log {
```

The `Log` *section has the same syntax as the* `Input` *section.*

```
    }
    Output {
```

The `Output` *section has the same syntax as the* `Input` *section.*

```
    }
    Configuration {
      File1 = String | StringReference;
    [ Path1 = String | StringReference; ]
    }
  } // end of section Simulation.Files
  [CallParameter {
    [Prefix = String | StringReference;]
    [Suffix = String | StringReference;]
  }]
  [ObjectiveFunctionLocation {
    Name1       = String;
    Delimiter1  = String | StringReference;  |  Function1  = String;

    [ Name2       = String;
      Delimiter2  = String | StringReference;  |  Function2  = String;
```

```
    [ ... ] ]

  }]
} // end of section Simulation
Optimization {
  Files {
    Command {
      File1 = String | StringReference;
    [ Path1 = String | StringReference; ]
    }
  }
} // end of section Optimization
```

The sections have the following meaning:

`Simulation.Files.Template` GenOpt writes the value of the independent variables to the simulation input files. To do so, GenOpt reads the simulation input *template* files, replaces each occurrence of `%variableName%` by the numerical value of the corresponding variable, and the resulting file contents are written as the simulation input files. The string `%variableName%` refers to the name of the variable as specified by the entry `Name` in the optimization command file on page 94.

The independent variables can be written to several simulation input files if required. To do so, specify as many `Filei` and `Pathi` assignments as necessary (where `i` stands for a one-based counter of the file and path name). Note that there must obviously be the same number of files and paths in the `Input` section that follows this section.

If there are multiple simulation input template files, each file will be written to the simulation input file whose keyword ends with the same number.

The following rules are imposed:

1. Each variable name specified in the optimization command file *must* occur in at least one simulation input template file or in at least one function that is specified in the section `ObjectiveFunctionLocation` below.
2. Multiple occurrences of the same variable name are allowed in the same file and in the same function specification (as specified by the keyword `Functioni`, `i` $= 1, 2, \ldots$).
3. If the value `WriteStepNumber` in the section `OptimizationSettings` of the optimization command file is set to `true`, then rule 1 and 2 apply also to `%stepNumber%`. If `WriteStepNumber` is set to `false`, then `%stepNumber%` can occur, but it will be ignored.

`Simulation.Files.Input` The simulation input file is generated by GenOpt based on the current parameter set and the corresponding simulation input *template* file, as explained in the previous paragraph. Obviously, the number of simulation input files must be equal to the number of simulation input template files.

The section `Input` has an optional keyword, called `SavePath`. If `SavePath` is specified, then the corresponding input file will after each simulation be copied into the directory specified by `SavePath`. The copied file will have the same name, but with the simulation number added as prefix.

`Simulation.Files.Log` GenOpt scans the simulation log file for error messages. The optimization terminates if any of the strings specified by the

variable `ErrorMessage` in the `SimulationError` section of the GenOpt configuration file is found. At least one simulation log file must be specified.

The section `Log` also has the optional keyword `SavePath`. It has the same functionality as explained in the previous section.

`Simulation.Files.Output` GenOpt reads the cost function value from this file. GenOpt assumes that the value that is written after the *last* occurrence of the string specified by `Delimiteri` (`i` $= 1, 2, \ldots$) in the section `ObjectiveFunctionLocation` is the cost function value. The number of cost function values is arbitrary (but at least one must be specified). The optimization algorithms minimize the first cost function value. The other values can be used for post-processing of the simulation output. They will also be reported to the output files and the online chart.

GenOpt searches for the cost function value as follows:

1. After the first simulation, GenOpt searches for the first cost function value in the first output file. The number that occurs after the *last* occurrence of the string specified by the variable `Delimiteri` (`i` $= 1, 2, \ldots$) in the section `ObjectiveFunctionLocation` is assumed to be the cost function value. If the first output file does not contain the first cost function value, then GenOpt reads the second output file (if present) and so on until the last output file is read. If GenOpt cannot find the cost function value in any of the output files or function definitions, it will terminate with an error. The same procedure is repeated with the second cost function value (if present) until all cost function values have been found.
2. In the following iterations, GenOpt will only read the file(s) where it found the cost function value(s) after the first simulation. The files that did not contain a cost function value after the first simulation will not be read anymore.

This section also contains the optional keyword `SavePath`. If this keyword is specified, then GenOpt copies the output file. This is particularly useful for doing parametric runs.

`Simulation.Files.Configuration` The entries in this section specify the simulation configuration file, which contains information that is related to the simulation program only, but not related to the optimization problem. The simulation configuration file is explained below.

`Simulation.CallParameter` Here, a prefix and suffix for the command that starts the simulation program can be added. With these entries, any additional information, such as the name of the weather file, can be passed to the simulation program. To do so, one has to refer to either of these entries in the argument of the keyword `Command` (see page 93).

`Simulation.ObjectiveFunctionLocation` This section specifies where the cost function values can be found in the simulation output files, and possibly how these values have to be post-processed before they will be passed to the optimization algorithm.

GenOpt reads the value after the *last* occurrence of `Delimiteri` (`i` $= 1, 2, \ldots$) as the cost function value. The value of `Namei` is used to label the results in the output reports.

Alternatively to the entry `Delimiteri`, an entry `Functioni` can be specified to define how the cost function values should be post-processed. See page 98 for an example.

For convenience, the section `ObjectiveFunctionLocation` can optionally be specified in the initialization file, but its specification is required in the configuration file. If this section is specified in both files, then the specification in the initialization file will be used.

Specifying the section `ObjectiveFunctionLocation` in the initialization file is of interest if a simulation program is used for different problems that require different values of this section. Then, the same (simulation program specific) configuration file can be used for all runs and the different settings can be specified in the (project dependent) initialization file rather than in the configuration file.

`Optimization.Files.Command` This section specifies where the optimization command file is located. This file contains the mathematical information of the optimization. See page 94 for a description of this file.

11.1.2 Configuration File

The configuration file contains information related only to the simulation program used and not to the optimization problem. Hence, it has to be written only once for each simulation program and operating system. We recommend to put this file in the directory `cfg` so that it can be used for different optimization projects. Some configuration files are provided with the GenOpt installation.

The syntax is specified by

```
// Error messages of the simulation program.
SimulationError{
   ErrorMessage = String;
  [ErrorMessage = String;
  [ ... ] ]
}

// Number format for writing simulation input files.
IO{
   NumberFormat = Float | Double;
}

// Specifying how to start the simulation program.
SimulationStart{
   Command = String;
   WriteInputFileExtension = Boolean;
}

// Specifying the location of the
// cost function value in the simulation output file
ObjectiveFunctionLocation{
    Name1       = String;
    Delimiter1  = String | StringReference;  |  Function1  = String;

    [ Name2       = String;
      Delimiter2  = String | StringReference;  |  Function2  = String;
    [ ... ] ]
}
```

The entries have following meaning:

`SimulationError` The error messages that might be written by the simulation program must be assigned to the keyword `ErrorMessage` so that GenOpt can check whether the simulation has completed successfully. At least one entry for `ErrorMessage` must be given.

`IO` The keyword `NumberFormat` specifies in what format the independent parameters will be written to the simulation input file. The setting `Double` is recommended, unless the simulation program cannot read this number format.

`SimulationStart` The keyword `Command` specifies what string must be used to start the simulation program. It is important that this command waits until the simulation terminates (see the directory `cfg` for examples). The value of the variable `Command` is treated in a special way: Any value of the optimization initialization file can be automatically copied into the

value of `Command`. To do so, surround the reference to the corresponding keyword with percent signs. For example, a reference to the keyword `Prefix` of the initialization file looks like

```
%Simulation.CallParameter.Prefix%
```

By setting `WriteInputFileExtension` to `false`, the value of the keyword `Simulation.Input.Filei` (where `i` stands for `1`, `2`, `3`) is copied into `Command`, and the file extension is removed.

`ObjectiveFunctionLocation` Note that this section can also be specified in the initialization file. The section in this file is ignored if this section is also specified in the configuration file. See page 92 for a description.

11.1.3 Command File

The command file specifies optimization-related settings such as the independent parameters, the stopping criteria and the optimization algorithm being used. The sequence of the entries in all sections of the command file is arbitrary.

There are two different types of independent parameters, *continuous parameters* and *discrete parameters*. Continuous parameters can take on any values, possibly constrained by a minimum and maximum value. Discrete parameters can take on only user-specified discrete values, to be specified in this file.

Some algorithms require all parameters to be continuous, or all parameters to be discrete, or allow both continuous and discrete parameters. Please refer to the algorithm section on page 15-73.

a) Specification of a Continuous Parameter

The structure for a continuous parameter is

```
// Settings for a continuous parameter
Parameter{
   Name = String;
   Ini  = Double;
   Step = Double;
   [ Min  = Double | SMALL; ]
   [ Max  = Double | BIG; ]
   [ Type = CONTINUOUS; ]
}
```

The entries are:

`Name` The name of the independent variable. GenOpt searches the simulation input template files for this string – surrounded by percent signs – and replaces each occurrence by its numerical value before it writes the simulation input files.

`Ini` Initial value of the parameter.

`Step` Step size of the parameter. How this variable is used depends on the optimization algorithm being used. See the optimization algorithm descriptions for details.

`Min` Lower bound of the parameter. If the keyword is omitted or set to `SMALL`, the parameter has no lower bound.

`Max` Upper bound of the parameter. If the keyword is omitted or set to `BIG`, the parameter has no upper bound.

`Type` Optional keyword that specifies that this parameter is continuous. By default, if neither `Type` nor `Values` (see below) are specified, then the parameter is considered to be continuous and the `Parameter` section must have the above format.

b) Specification of a Discrete Parameter

For discrete parameters you need to specify the set of admissible values. Alternatively, if a parameter is spaced either linearly or logarithmically, specify the minimum and maximum value of the parameter and the number of intervals.

First, we list the entry for the case of specifying the set of admissible values:

```
// Settings for a discrete parameter
Parameter{
   Name   = String;
   Ini    = Integer;
   Values = String;
 [ Type   = SET; ]
}
```

The entries are:

`Name` *As for the continuous parameter above.*

`Ini` 1-based index of the initial value. For example, if `Values` specifies three admissible values, then `Ini` can be either `1`, `2`, or `3`.

`Values` Set of admissible values. The entry must be of the form

```
Values = "value1, value2, value3";
```

i.e., the values are separated by a comma, and the list is enclosed in apostrophes (`"`). For `value1`, `value2`, etc., numbers and strings are allowed.

If all entries of `Values` are numbers, then the result reports contain the actual values of this entry. Otherwise, the result reports contain the index of this value, i.e., `1` corresponds to `value1`, `2` corresponds to `value2`, etc.

`Type` Optional keyword that specifies that this parameter is discrete. By default, if the entry `Values` is specified, a parameter is considered to be discrete, and the `Parameter` section must have the above format.

To obtain linear or logarithmic spacing between a minimum and maximum value, the `Parameter` section can be specified as

```
// Settings for a discrete parameter, linearly or logarithmically spaced
Parameter{
   Name = String;
   Ini  = Integer;
   Type = SET;
   Min  = Double;
   Max  = Double;
   Step = Integer;
}
```

`Name` *As for the continuous parameter above.*

`Ini` 1-based index of the initial value. For example, if `Step` is set to +2 or to −2, then `Ini` can be set to any integer between 1 and 3.

`Type` This variable must be equal to `SET`.

`Min` Minimum value of the spacing.

`Max` Maximum value of the spacing.

`Step` Number of intervals. If `Step` < 0, then the spacing is logarithmic, otherwise it is linear. Set `Step` $= 0$ to keep the parameter always fixed on its minimum value.

The linear or logarithmic spacing is computed using (7.1) on page 71.

c) Specification of Input Function Objects

The specification of input function objects in optional. If any input function object is specified, then its name must appear either in another input function object, in a simulation input template file, or in an output function object. Otherwise, GenOpt terminates with an error message. See Section 11.2 on page 98 for an explanation of input and output function objects.

The syntax for input function objects is

```
// Input function objects entry
Function{
   Name     = String;
   Function = String;
}
```

The entries are

`Name` A unique name that is not used for any other input function object and for any other independent parameter.

`Function` A function object (see Section 11.2 on page 98). The string must be enclosed by apostrophes (`"`).

d) Structure of the Command File

Using above structures of the `Parameter` section, the command file has the structure

```
// Settings of the independent parameters
Vary{
   // Parameter entry
     List any of the Parameter sections as described
     in the Sections 11.1.3.a) and 11.1.3.b).

   // Input function object
     List any of the Function sections as described
     in the Section 11.1.3.c).
}

// General settings for the optimization process
OptimizationSettings{
   MaxIte          = Integer;
   WriteStepNumber = Boolean;
 [ MaxEqualResults = Integer; ]
}

// Specification of the optimization algorithm
Algorithm{
   Main = String;
   ... // any other entries that are required
       // by the optimization algorithm
}
```

The different sections are:

`Vary` This section contains the definition of the independent parameter and the input function objects. See Sections 11.1.3.a), 11.1.3.b), and 11.1.3.c) for possible entries.

`OptimizationSettings` This section specifies general settings of the optimization. `MaxIte` is the maximum number of iterations. After `MaxIte` main iterations, GenOpt terminates with an error message.
`WriteStepNumber` specifies whether the current step of the optimization has to be written to the simulation input file or to a function object. The step number can be used to calculate a penalty or barrier function (see Section 8.2 on page 75).
The optional parameter `MaxEqualResults` specifies how many times the cost function value can be equal to a value that has previously been obtained before GenOpt terminates. This setting is used to terminate GenOpt if the cost function value is constant for several iterates (see Section 11.3). The default value of `MaxEqualResults` is 5.

`Algorithm` The setting of `Main` specifies which algorithm is invoked for doing the optimization. Its value has to be equal to the class name that contains the algorithm. Note that additional parameters might be required depending on the algorithm used (see Section 5 for the implemented algorithms).

11.1.4 Log File

GenOpt writes a log file to the directory that contains the initialization file. The name of the log file is `GenOpt.log`.

The GenOpt log file contains general information about the optimization process. It also contains warnings and errors that occur during the optimization.

11.1.5 Output File

In addition to `GenOpt.log`, GenOpt writes two output files to the directory where the optimization command file is located. (The location of the optimization command file is defined by the variable `Optimization.Files.Command.Path1` in the optimization initialization file).

The names of the output files are `OutputListingMain.txt` and `OutputListingAll.txt`. They list the optimization steps. `OutputListingMain.txt` contains only the main iteration steps, and `OutputListingAll.txt` contains all iteration steps.

Each time the method `genopt.algorithm.Optimizer.report()` is called from the optimization algorithm, the current trial is reported in either one of the files.

11.2 Pre-Processing and Post-Processing

Some simulation programs do not have the capability to pre-process the independent variables, or to post-process values computed during the simulation. For such situations, GenOpt's *input function objects* and *output function objects* can be used.

a) Function Objects

Function objects are formulas whose arguments can be the independent variables, the keyword `stepNumber`, and for post-processing, the result of the simulation.

Following functions are implemented:

Function	Returns
`add(x0, x1)`	$x^0 + x^1$
`add(x0, x1, x2)`	$x^0 + x^1 + x^2$
`subtract(x0, x1)`	$x^0 - x^1$
`multiply(x0, x1)`	$x^0 x^1$
`multiply(x0, x1, x2)`	$x^0 x^1 x^2$
`divide(x0, x1)`	x^0 / x^1
`log10(x0)`	$\log_{10}(x^0)$

Furthermore, all functions that are defined in the class `java.lang.StrictMath` and whose arguments and return type are of type `double` can be accessed by typing their name (without the package and class name).

In addition, users can implement any other static method with arguments and return type `double` by adding the method to the file `genopt/algorithm/util/math/Fun.java`. The method must have the syntax

```
public static double  methodName(double x0, double x1) {
  double r;
  //  do any computations
  return r;
}
```

The number of arguments is arbitrary.

Compile the file after adding new methods. No other changes are required. To compile the file, a Java compiler must be installed (such as the one from Sun Microsystems). To compile it, open a console (or DOS window), change to the directory `genopt/algorithm/util/math` and type

```
javac -source 1.4 Fun.java
```

This will generate the file `Fun.class`. If the compilation fails, then the variable `CLASSPATH` is probably not set as described in Chapter 10.

Next, we present an example for pre-processing and afterwards an example for post-processing.

b) Pre-Processing

Example 11.2.1 Suppose we want to find the optimal window width and height. Let `w` and `h` denote the window width and height, respectively. Suppose we want the window height to be 1/2 times the window width, and the window width must be between 1 and 2 meters. Then, we could specify in the command file the section

```
Parameter{
   Name =   w;
   Ini  = 1.5;     Step = 0.05;
   Min  =   1;     Max  =    2;
```

```
   Type = CONTINUOUS;
}
Function{
   Name =   h;     Function = "multiply( %w%, 0.5 )";
}
```

Then, in the simulation input template files, GenOpt will replace all occurrences of `%w%` by the window width and all occurences of `%h%` by 1/2 times the numerical value of `%w%`. □

GenOpt does not report values that are computed by input functions. To report such values, they need to be specified in the section `ObjectiveFunctionLocation`, as shown in Example 11.2.2 below.

c) Post-Processing

Example 11.2.2 Suppose we want to minimize the sum of annual heating and cooling energy consumption, which we will call *total energy*. Some simulation programs cannot add different output variables. For example, EnergyPlus [CLW+01] writes the heating and cooling energy consumption separately to the output file. In order to optimize the total energy, the simulation output must be post-processed.

To post-process the simulation results in GenOpt, we can proceed as follows:
Suppose the cost function delimiter (see Section 11.1.1) for the heating and cooling energy are, respectively, `Eheat=` and `Ecool=`. In addition, suppose we want to report the value of the variable `h` that has been computed by the input function object in Example 11.2.1.

Then, in the optimization initialization file (see Section 11.1.1) we can specify the section

```
ObjectiveFunctionLocation{
   Name1 = E_tot;  Function1 = "add( %E_heat%, %E_cool% )";
   Name2 = E_heat; Delimiter2 = "Eheat=";
   Name3 = E_cool; Delimiter3 = "Ecool=";
   Name4 = height; Function4  = %h%;
}
```

This specification causes GenOpt to (i) substitute the value of `h` in `Function4`, (ii) read from the simulation output file(s) the numbers that occur after the strings `Eheat=` and `Ecool=`, (iii) substitute these numbers into the function `add( %E_heat%, %E_cool% )`, (iv) evaluate the functions `Function1` and `Function4`, and (v) minimize the sum of heating and cooling energy. □

As arguments of the function defined above, we can use any name of an independent variable, of an input function object, or the keyword `%stepNumber%`.

11.3 Truncation of Digits of the Cost Function Value

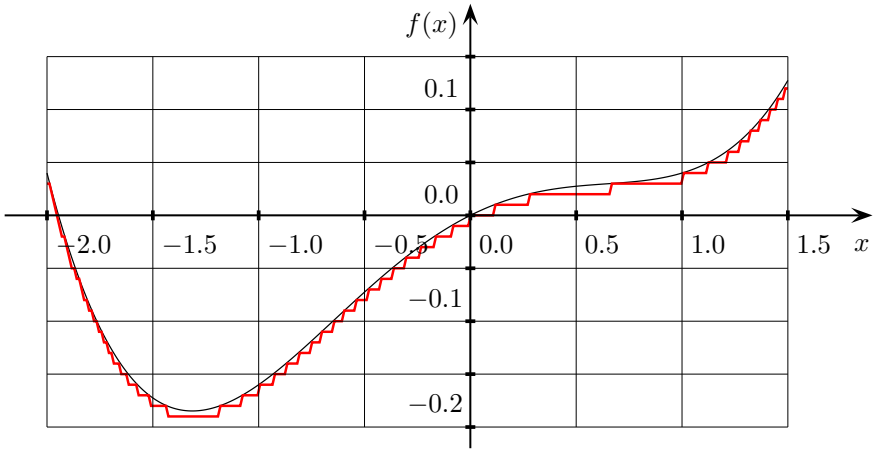


Figure 11.1: Function (11.1) *with machine precision and with truncated digits. The upper line shows the cost function value with machine precision, and the lower line shows the cost function value with only two digits beyond the decimal point.*

For $x' \in \mathbb{R}^n$ and $f\colon \mathbb{R}^n \to \mathbb{R}$, assume there exists a scalar $\delta > 0$ such that $f(x') = f(x'')$ for all $x'' \in B(x', \delta)$, where $B(x', \delta) \triangleq \{x'' \in \mathbb{R}^n \mid \|x' - x''\| < \delta\}$. Obviously, in $B(x', \delta)$, an optimization algorithm can fail because iterates in $B(x', \delta)$ contain no information about descent directions outside of $B(x', \delta)$. Furthermore, in absence of convexity of $f(\cdot)$, the optimality of x' cannot be ascertain in general.

Such situations can be generated if the simulation program writes the cost function value to the output file with only a few digits. Fig. 11.1 illustrates that truncating digits can cause problems particularly in domains of $f(\cdot)$ where the slope of $f(\cdot)$ is flat. In Fig. 11.1, we show the function

$$f(x) \triangleq 0.1\,x - 0.1\,x^2 + 0.04\,x^4. \tag{11.1}$$

The upper line is the exact value of $f(\cdot)$, and the lower line is the rounded value of $f(\cdot)$ such that it has only two digits beyond the decimal point. If the optimization algorithm makes changes in x in the size of 0.2, then it may fail for $0.25 < x < 1$, which is far from the minimum. In this interval, no useful information about the descent of $f(\cdot)$ can be obtained. Thus, the cost function must be written to the output file without truncating digits.

To detect such cases, the optimization algorithm can cause GenOpt to check whether a cost function values is equal to a previous cost function value. If the same cost function value is obtained more than a user-specified number of times, then GenOpt terminates with an error message. The maximum number

of equal cost function values is specified by the parameter `MaxEqualResults` in the command file (see page 94).

GenOpt writes an information to the user interface and to the log file if a cost function value is equal to a previous function value.

12 Conclusion

In system optimization it is not possible to apply a general optimization algorithm that works efficiently on all problems. What algorithm should be used depends on the properties of the cost function, such as the number of independent parameters, the continuity of the cost function and its derivatives, and the existence of local minima. Thus a variety of optimization algorithms is needed. To address this need, GenOpt has a library with different optimization algorithms and an optimization algorithm interface that users can use to implement their own optimization algorithm if desired.

For optimizing the the functions that GenOpt is aimed at, a generalization of the structure of the optimization process can be made. The fact that analytical properties of the cost function are unavailable makes it possible to separate optimization and function evaluation. Therefore, GenOpt has a simulation program interface that allows coupling any program that exchanges input and output using text files. Hence, users are not restricted to using a special program for evaluating the cost function. Rather, they can use the simulation program they are already using for their system design and development. Thus, the system can be optimized with little additional effort.

This open environment not only allows coupling any simulation program and implementing special purpose algorithms, but it also allows sharing algorithms among users. This makes it possible to extend the algorithm library and thus extend GenOpt's applicability.

13 Acknowledgment

The development of GenOpt was sponsored by grants from the Swiss Academy of Engineering Sciences (SATW), the Swiss National Energy Fund (NEFF) and the Swiss National Science Foundation (SNF) and is supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, Office of Building Technology Programs of the U.S. Department of Energy, under Contract No. DE-AC03-76SF00098. I would like to thank these institutions for their generous support.

14 Notice

The Government is granted for itself and others acting on its behalf a paid-up, nonexclusive, irrevocable, worldwide license in this data to reproduce, prepare derivative works, and perform publicly and display publicly. Beginning five (5) years after the date permission to assert copyright is obtained from the U.S. Department of Energy, and subject to any subsequent five (5) year renewals, the Government is granted for itself and others acting on its behalf a paid-up, nonexclusive, irrevocable, worldwide license in this data to reproduce, prepare derivative works, distribute copies to the public, perform publicly and display publicly, and to permit others to do so. NEITHER THE UNITED STATES NOR THE UNITED STATES DEPARTMENT OF ENERGY, NOR ANY OF THEIR EMPLOYEES, MAKES ANY WARRANTY, EXPRESS OR IMPLIED, OR ASSUMES ANY LEGAL LIABILITY OR RESPONSIBILITY FOR THE ACCURACY, COMPLETENESS, OR USEFULNESS OF ANY INFORMATION, APPARATUS, PRODUCT, OR PROCESS DISCLOSED, OR REPRESENTS THAT ITS USE WOULD NOT INFRINGE PRIVATELY OWNED RIGHTS.

A Benchmark Tests

This section lists the settings used in the benchmark tests on page 61.

The settings in `OptimizationsSettings` and `Algorithm` are the same for all runs expect for `Accuracy`, which is listed in the result chart on page 62.

The common settings were:

```
OptimizationSettings{
    MaxIte          = 1500;
    WriteStepNumber = false;
}

Algorithm{
   Main = NelderMeadONeill;
   Accuracy =  see page 62;
   StepSizeFactor = 0.001;
   BlockRestartCheck = 5;
   ModifyStoppingCriterion  =  see page 62;
}
```

The benchmark functions and the `Parameter` settings in the `Vary` section are shown below.

A.1 Rosenbrock

The Rosenbrock function that is shown in Fig A.1 is defined as

$$f(x) \triangleq 100\left(x^2 - (x^1)^2\right)^2 + (1 - x^1)^2, \tag{A.1}$$

where $x \in \mathbb{R}^2$. The minimum is at $x^* = (1, 1)$, with $f(x^*) = 0$.

The section `Vary` of the optimization command file was set to

```
Vary{
   Parameter{
      Name =   x1; Min = SMALL;
      Ini  = -1.2; Max = BIG;
      Step =    1;
      }
   Parameter{
      Name =   x2; Min = SMALL;
      Ini  =    1; Max = BIG;
      Step =    1;
      }
}
```

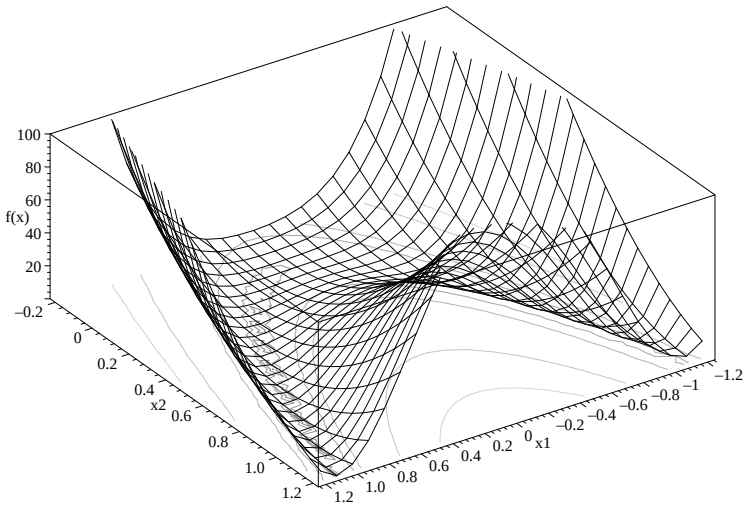


Figure A.1: *Rosenbrock function.*

A.2 Function 2D1

This function has only one minimum point. The function is defined as

$$f(x) \triangleq \sum_{i=1}^{3} f^i(x), \tag{A.2}$$

with

$$f^1(x) \triangleq \langle b,\, x\rangle + \frac{1}{2}\langle x,\, Q\, x\rangle, \quad b \triangleq \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \quad Q \triangleq \begin{pmatrix} 10 & 6 \\ 6 & 8 \end{pmatrix}, \tag{A.3}$$

$$f^2(x) \triangleq 100 \arctan\big((2 - x^1)^2 + (2 - x^2)^2\big), \tag{A.4}$$

$$f^3(x) \triangleq 50 \arctan\big((0.5 + x^1)^2 + (0.5 + x^2)^2\big), \tag{A.5}$$

where $x \in \mathbb{R}^2$. The function has a minimum at $x^* = (1.855340, 1.868832)$, with $f(x^*) = -12.681271$. It has two regions where the gradient is very small (see Fig. A.2).

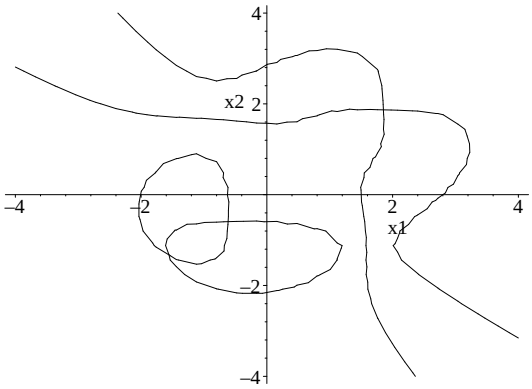


Figure A.2: Contour plot of $\frac{\partial df(x)}{\partial x^1} = 0$ and $\frac{\partial df(x)}{\partial x^2} = 0$, where $f(x)$ is as in (A.2).

The section `Vary` of the optimization command file is

```
Vary{
   Parameter{
      Name = x0; Min = SMALL;
      Ini  = -3; Max = BIG;
      Step = 0.1;
   }
   Parameter{
      Name =  x1; Min = SMALL;
      Ini  =  -3; Max = BIG;
      Step = 0.1;
   }
}
```

A.3 Function Quad

The function "Quad" is defined as

$$f(x) \triangleq \langle b,\, x\rangle + \frac{1}{2}\langle x,\, M\, x\rangle, \tag{A.6}$$

where $b, x \in \mathbb{R}^{10}$, $M \in \mathbb{R}^{10\times 10}$, and

$$b \triangleq (10,\, 10,\, \ldots\, ,\, 10)^T. \tag{A.7}$$

This function is used in the benchmark test with two different positive definite matrices M. In one test case, M is the identity matrix I and in the other test case M is a matrix, called Q, with a large range of eigenvalues. The matrix Q has elements

579.7818	−227.6855	49.2126	−60.3045	−152.4101	−207.2424	8.0917	33.6562	204.1312	−3.7129
−227.6855	236.2505	−16.7689	−40.3592	179.8471	80.0880	−64.8326	15.2262	−92.2572	40.7367
49.2126	−16.7689	84.1037	−71.0547	20.4327	5.1911	−58.7067	−36.1088	−62.7296	7.3676
−60.3045	−40.3592	−71.0547	170.3128	−140.0148	8.9436	26.7365	125.8567	62.3607	−21.9523
−152.4101	179.8471	20.4327	−140.0148	301.2494	45.5550	−31.3547	−95.8025	−164.7464	40.1319
−207.2424	80.0880	5.1911	8.9436	45.5550	178.5194	22.9953	−39.6349	−88.1826	−29.1089
8.0917	−64.8326	−58.7067	26.7365	−31.3547	22.9953	124.4208	−43.5141	75.5865	−32.2344
33.6562	15.2262	−36.1088	125.8567	−95.8025	−39.6349	−43.5141	261.7592	86.8136	22.9873
204.1312	−92.2572	−62.7296	62.3607	−164.7464	−88.1826	75.5865	86.8136	265.3525	−1.6500
−3.7129	40.7367	7.3676	−21.9523	40.1319	−29.1089	−32.2344	22.9873	−1.6500	49.2499

The eigenvalues of Q are in the range of 1 to 1000.

The functions have minimum points x^* at

Matrix M:	I	Q
x^{*^0}	−10	−2235.1810
x^{*^1}	−10	−1102.4510
x^{*^2}	−10	790.6100
x^{*^3}	−10	−605.2480
x^{*^4}	−10	−28.8760
x^{*^5}	−10	228.7640
x^{*^6}	−10	−271.8830
x^{*^7}	−10	−3312.3890
x^{*^8}	−10	−2846.7870
x^{*^9}	−10	−718.1490
$f(x^*)$	−500	0

Both test functions have been optimized with the same parameter settings. The settings for the parameters `x0` to `x9` are all the same, and given by

```
Vary{
   Parameter{
      Name = x0; Min = SMALL;
      Ini  =  0; Max = BIG;
      Step =  1;
   }
}
```

Bibliography

[AD03] Charles Audet and J. E. Dennis, Jr. Analysis of generalized pattern searches. *SIAM Journal on Optimization*, 13(3):889–903, 2003.

[Avr76] Mordecai Avriel. *Nonlinear programming*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1976.

[BP66] M. Bell and M. C. Pike. Remark on algorithm 178. *Comm. ACM*, 9:685–686, September 1966.

[CD01] A. Carlisle and G. Dozier. An off-the-shelf PSO. In *Proceedings of the Workshop on Particle Swarm Optimization*, Indianapolis, IN, 2001.

[CK02] Maurice Clerc and James Kennedy. The particle swarm – explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6(1):58–73, February 2002.

[Cla90] F. H. Clarke. *Optimization and nonsmooth analysis*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1990.

[CLW+01] Drury B. Crawley, Linda K. Lawrie, Frederick C. Winkelmann, W.F. Buhl, Y. Joe Huang, Curtis O. Pedersen, Richard K. Strand, Richard J. Liesen, Daniel E. Fisher, Michael J. Witte, and Jason Glazer. Energyplus: creating a new-generation building energy simulation program. *Energy and Buildings*, 33(4):443–457, 2001.

[CP00] I. D. Coope and C. J. Price. Positive bases in numerical optimization. Technical Report UCDMS2000/12, Dept. of Mathematics and Statistics, Univ. of Canterbury, Christchurch, New Zealand, 2000.

[Dav54] Chandler Davis. Theory of positive linear dependence. *American Journal of Mathematics*, 76(4):733–746, October 1954.

[DV68] R. De Vogelaere. Remark on algorithm 178. *Comm. ACM*, 11:498, July 1968.

[EK95] R. C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Sixth International Symposium on Micro Machine and Human Science*, pages 39–43, Nagoya, Japan, October 1995. IEEE.

[ES01] R. C. Eberhart and Y. Shi. Particle swarm optimization: Developments, applications and resources. In *Proceedings of the 2001 Congress on Evolutionary Computation*, volume 1, pages 84–86, Seoul, South Korea, 2001. IEEE.

[HJ61] R. Hooke and T. A. Jeeves. 'Direct search' solution of numerical and statistical problems. *J. Assoc. Comp. Mach.*, 8(2):212–229, 1961.

[KDB76] S. A. Klein, J. A. Duffie, and W. A. Beckman. TRNSYS – A transient simulation program. *ASHRAE Transactions*, 82, 1976.

[KE95] J. Kennedy and R. C. Eberhart. Particle swarm optimization. In *IEEE International Conference on Neural Networks*, volume IV, pages 1942–1948, Perth, Australia, November 1995.

[KE97] J. Kennedy and R. C. Eberhart. A discrete binary version of the particle swarm algorithm. In *Proc. of Systems, Man, and Cybernetics*, volume 5, pages 4104–4108. IEEE, October 1997.

[KES01] James Kennedy, Russell C. Eberhart, and Yuhui Shi. *Swarm Intelligence*. Morgan Kaufmann Publishers, 2001.

[KM02] J. Kennedy and R. Mendes. Population structure and particle swarm performance. In David B. Fogel, Mohamed A. El-Sharkawi, Xin Yao, Garry Greenwood, Hitoshi Iba, Paul Marrow, and Mark Shackleton, editors, *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, pages 1671–1676. IEEE, 2002.

[LPV02a] E. C. Laskari, K. E. Parsopoulos, and M. N. Vrahatis. Particle swarm optimization for integer programming. In *Proceedings of the 2002 Congress on Evolutionary Computation*, volume 2, pages 1582–1587, Honolulu, HI, May 2002. IEEE.

[LPV02b] E. C. Laskari, K. E. Parsopoulos, and M. N. Vrahatis. Particle swarm optimization for minimax problems. In *Proceedings of the 2002 Congress on Evolutionary Computation*, volume 2, pages 1576–1581, Honolulu, HI, May 2002. IEEE.

[LRWW98] Jeffrey C. Lagarias, James A. Reeds, Margaret H. Wright, and Paul E. Wright. Convergence properties of the Nelder-Mead simplex method in low dimensions. *SIAM Journal on Optimization*, 9(1):112–147, 1998.

[McK98] K. I. M. McKinnon. Convergence of the Nelder-Mead simplex method to a nonstationary point. *SIAM Journal on Optimization*, 9(1):148–158, 1998.

[NM65] J. A. Nelder and R. Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, January 1965.

[O'N71] R. O'Neill. Algorithm AS 47 – Function minimization using a simplex procedure. *Appl. Stat. 20*, 20:338–345, 1971.

[PFTV93] W. H. Press, B. P. Flannery, S. A. Tuekolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing*, chapter 20. Cambridge University Press, 1993.

[Pol97] Elijah Polak. *Optimization, Algorithms and Consistent Approximations*, volume 124 of *Applied Mathematical Sciences*. Springer Verlag, 1997.

[PV02a] K. E. Parsopoulos and M. N. Vrahatis. Particle swarm optimization method for constrained optimization problems. In P. Sincak, J. Vascak, V. Kvasnicka, and J. Pospichal, editors, *Intelligent Technologies – Theory and Applications: New Trends in Intelligent Technologies*, volume 76 of *Frontiers in Artificial Intelligence and Applications*, pages 214–220. IOS Press, 2002. ISBN: 1-58603-256-9.

[PV02b] K. E. Parsopoulos and M. N. Vrahatis. Recent approaches to global optimization problems through Particle Swarm Optimization. *Natural Computing*, 1:235–306, 2002.

[PW03] Elijah Polak and Michael Wetter. Generalized pattern search algorithms with adaptive precision function evaluations. Technical Report LBNL-52629, Lawrence Berkeley National Laboratory, Berkeley, CA, 2003.

[SE98] Y. Shi and R. C. Eberhart. A modified particle swarm optimizer. In *Evolutionary Computation*, IEEE World Congress on Computational Intelligence, pages 69–73, Anchorage, AK, May 1998. IEEE.

[SE99] Y. Shi and R. C. Eberhart. Empirical study of particle swarm optimization. In *Proceedings of the 1999 Congress on Evolutionary Computation*, volume 3, pages 1945–1950, Carmel, IN, 1999. IEEE.

[Smi69] Lyle B. Smith. Remark on algorithm 178. *Comm. ACM*, 12:638, November 1969.

[Tor97] Virginia Torczon. On the convergence of pattern search algorithms. *SIAM Journal on Optimization*, 7(1):1–25, 1997.

[vdBE01] F. van den Bergh and A.P Engelbrecht. Effects of swarm size on cooperative particle swarm optimisers. In *GECCO*, San Francisco, CA, July 2001.

[Wal75] G. R. Walsh. *Methods of optimization*. Wiley-Interscience [John Wiley & Sons], London, 1975.

[WBB+93] F.C. Winkelmann, B. E. Birsdall, W. F. Buhl, K. L. Ellington, A. E. Erdem, J. J. Hirsch, and S. Gates. *DOE-2 Supplement, Version 2.1E*. Lawrence Berkeley National Laboratory, Berkeley, CA, USA, November 1993.

[Wil64] D. J. Wilde. *Optimum seeking methods*. Prentice-Hall, USA, 1964.

[WP03] Michael Wetter and Elijah Polak. A convergent optimization method using pattern search algorithms with adaptive precision simulation. In *To appear: Proc. of the 8-th IBPSA Conference*, Eindhoven, NL, August 2003.

[Wri96] M. H. Wright. Direct search methods: once scorned, now respectable. In D. F. Griffiths and G. A. Watson, editors, *Numerical Analysis 1995*, pages 191–208. Addison Wesley Longman (Harlow), 1996.